1856

CYBERNETIC MUSIC

JAXITRON



CYBERNETIC MUSIC

JAXITRON


TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

FIRST EDITION
FIRST PRINTING

Library of Congress Cataloging in Publication Data

Jaxitron.
Cybernetic music.

Includes index.
1. Computer composition. 2. Computer music—Instruction and study. 3. Cybernetics. I. Title.
MT56.J4 1985 781.6'1'0285 85-17305

ISBN 0-8306-0856-7
ISBN 0-8306-1856-2 (pbk.)

Contents

Introduction

The subject of this book is computer *music*, not *computer* music. Though it may seem fitting to make this distinction through the use of intonation, intonation in words or music helps convey information esthetically or emotionally—but not conceptually. "Cybernetic" music is a more appropriate characterization, but that term is not so commonly used that it can stand without further definition.

Before explaining (in the next chapter) exactly how that word applies to our subject, I will surprise no one by saying that it does indeed have something to do with computers. It also has much to do with the human mind, but then so do musical composition and computer programming. Others feel my subject belongs under the heading, "computer-aided composition," but this strikes me as misrepresenting the leading role of the mind in the entire process. We are not going to play regurgitant games, feeding the computer random numbers to see what it comes up with. Nor will we connect eletrodes to our scalps and will the computer to make pretty sounds, as in certain biofeedback experiments labeled "cybernetic music"!

We are going to impose the logic of music on computers so that computers can help us compose. We will also expose music to the logic of computing to free music from unreasonably outdated and inhibiting lines of thought. Computers can help in composition by performing quite routine tasks such as keeping track of the time durations allotted to each measure and transposing a sequence of pitches through a given interval. But we'll see that they can even compose melodies and harmonize them just as we would ourselves—that is, just as we would if the methods we describe to the computer really reflect our own methods. The key to *Cybernetic Music* is the description of method itself.

Standard musical notation and terminology may be used to demonstrate method, but they cannot describe it. If we are to explain it to a computer, we must certainly be able to describe it to ourselves. Anyone who has ever listened to a composer trying to explain just what it is that he does will ap-

preciate the difficulties of part of this task. Anyone who has ever tried to get a computer to follow his wishes—not his programmed statements—will appreciate the remaining difficulties.

By approaching composition from such fundamental levels, readers with some background in music but none in computing will be on equal footing here with those possessing some computer experience but no musical training. Ideas from the two fields will intermingle freely and quite unconventionally, so that aptitude and interest will be the only prerequisites for newcomers to both areas—although a first course in a computer language, such as BASIC, and a few piano or guitar lessons would be most helpful. Younger readers relatively free from traditional cultural views that segregate the arts and sciences will probably fare best. Of course, there is no way that a book can provide hands-on computer experience or ears-on music experience. For those, I encourage actual use of the printed examples. But this brings us to another limitation: individual tastes in music.

Not without trepidation, I will try to present most musical examples in "unoffensive" styles. That is to say, to forestall that embarrassing question, "But can computers help write *real* music—you know, *good* music?" I will be most presumptuous in forcing these adjectives to be interpreted in a culturally averaged sense, thereby hoping to reach an acceptable middle region of the stylistic spectrum. As with all averages, this one may have little relation to any individual taste, my own included. Perhaps a bit facetiously, I find myself calling this musical style "the MT idiom," intending reference to Musical Theater rather than to the homologous-sounding word some would say measures the intellectual content of such music.

Of course, there are those who will play the results shown in the examples, disregard my "alibi" concerning cultural averages, and then take noticeable delight in repeating that question. To answer them, it would be necessary to program the methods of their favorite composers. Oddly enough, that would be a simpler task than the one that will concern us here. The existing works of a composer no longer living comprise a finite and well-defined target for analysis. To extend such a repertoire pseudo-cybernetically for the environment of those desiring to live in the past may be lucrative, but it borders on the macabre. Just why such analysis is easier than the task that lies ahead will be made clear shortly with the extended definition of what I mean by "cybernetic music."

Academically, musical composition has never been considered a subject that can be absorbed without prerequisite training in the elements of music. Computer programming, while a far younger subject, has been thought of in the same restrictive way, except that the prerequisites lay in the mathematical area. Like bumblebees unaware of academic pronouncements that they can't fly, many of the "uninitiated" continually undermine the complacency of academicians by soaring to new heights in both fields. I would like to think that it is with such spirits in mind and not with foolhardiness that I attempt here to present fundamentals in both subjects at once. More realistically, I consider this a book on musical composition in which a new tool with revolutionary impact on the topic must be covered. In fact, it is because of this tool that the ideas contained here can now be expressed in a way that is not purely academic.

We will take up a fairly detailed overview of a specific computer programming language, APL. That material will then serve as ostinato (i.e., it will persistently reappear as a background) in all that follows. The traditional musical components—rhythm, melody, and harmony—and their combinations will then be subjected to a most untraditional examination.

Such contrapuntal mixing of tradition and unorthodoxy introduces overtones that will not reverberate satisfactorily for all readers at all times. New approaches are required to take advantage of powerful new tools, whether or not they can produce acceptable substitutes for traditional products. It is well known that computers can produce new products which, in the case of music, are not substitutes and, for many listeners, are not even acceptable. As new hardware appears, making musical experimentation increasingly available to one and all but adding little to the supply of musical

perspective, traditional styles of music will form a vital link between the new methods and most experimenters' past listening experiences. Unusual methods will spontaneously suggest new lines of musical thought, and it would be as shameful to shelter the innocent from such ideas as it would be to cause the sophisticated to think that the new methods are merely newfangled ways of doing the same old thing. So at times I will stray ever so slightly from the MT path, hoping to suggest fruitful directions for experimentation but not to encourage making musical mayhem.

In composing, the mind frequently distracts itself with worrisome questions about the correct use of method and even the philosophical base for a given method. This is particularly true with new composers and with experienced ones writing in new ways. It is customary in teaching music to present the rules not as axioms but as dogma. Presumably the student will then not be bothered by deep soul searching, but only by questions of validity, i.e., "right" or "wrong." This is not a tradition I wish to follow.

There is much that could be said about the Artistic Method and the philosophy behind it. A discussion of the evolutionary principles that determine which of our behavioral traits are species-specific, which are individual though still hereditary, and which are learned can also shed interesting light on our emotional responses to good and bad, right and wrong. But my editor has wisely pointed out that such discussions would be misplaced in a book of this sort. So I will just say here that each of my methods should only be considered "a method" and not "The Method." The subjective reasoning behind my approach will not always be given here. Only in those instances where an objective technique cries out for some explanation of the underlying rationale will I venture into the shallows of those murky waters.



1 The Cybernetics of Music

According to my *Webster's Third New International Dictionary*, the word *cybernetics* derives from the Greek *kybernetes*, meaning "steersman" or "governor." It then goes on to say that cybernetics is a study in which the automatic control system made up of the human brain and nerve complex is compared with control systems designed around (inorganic) electromechanical devices. No mention is made of any of the possible motives one might have for conducting such a study, and, even if reasons were given, the chance of finding musical composition listed seems somewhat remote.

Well, suppose we decided to undertake such a study, comparing a composer with a computer for purposes somehow related to musical composition. The more vaguely we state our purpose, the further our investigation might range. That is, both systems have physical and chemical structure, both transmit electrical signals through a complex maze of circuitry, and both process information presumably related to the subject we have in mind. Merely mentioning musical composition in connection with our purpose, however, should be sufficient to steer our thoughts (cybernetics?) in the direction of information and its processing.

If we could describe in enough detail the way a composer manipulates information to produce a score, we could certainly reprogram the process for a computer. Before anyone improperly paraphrases what I just said, please note that, despite my use of the indefinite article, we're considering *the* composer's own information being manipulated by his or her own methods. How "we" could come into possession of "enough detail" to write such a program is not suggested. The fact that any living composer is also a learning composer, and thus a composer whose information and methods are constantly changing, makes it unlikely that anyone but that composer himself could carry out such a study while he lives. We are talking about *his* governship, *his* steering in treating all that information; *he* is the *keybernetes*.

A primary goal of all composers is to write in an original way, no matter how hackneyed the

idiom. No one interested in composing would care to program a computer to write like "a" composer. On hearing my claim about the possibility of writing a program that could compose a score, "the" composer would immediately ask, "Do you mean that if I could describe in detail how I would like to manipulate certain musical elements, I could get a computer to do it for me?" That is indeed what I said, but in a less personal way. And there are a number of other implied corollaries which go a long way toward answering the next question, "But why would I even want to?"

No matter what your level of musical sophistication, you can write a program to mimic the way you *intend* to treat given material. Effectively then, you can ask the computer what *your* music would be like if you did such and such. There has never been so powerful an instructor or decision-making tool for helping you determine which methods have promise and which do not. As you gain confidence in your style, you can delegate the well-defined parts of your composing process to a computer, but have it stop to ask you for additional information whenever necessary. There are at least two excellent reasons why you will want to work in this interactive way.

First, in your lifetime you will not be able to (or even want to) define your style in "final" form; while you live, your mind is yours to change and pack with new information. Composers often extend their methods until the "recipe" becomes too tedious to follow through to completion. The enjoyment of devising involved methods is canceled by the drudgery needed to carry them out. With a computer performing the labor, the *composition* of method can continue to grow and evolve. The excitement of composing, as in any other activity, is most intense near the frontier of your musical thinking. With the barrier of toil removed, your creative energies can focus on this area.

And second, there will always be places in your logic where the next step depends on the outcome of the previous steps. That is, you will need to hear or see what has been produced up to this point before deciding how to proceed. Even U-turns should be permitted, to let you respecify the choice that had been made at the Nth beat in the Mth bar. As the discipline called "artificial intelligence" matures, the computer may be able to go further without assistance, but our methods will become correspondingly more thoughtful, keeping alive and even increasing the need for human intervention.

There are also corollaries that will gain in appeal as you acquire musical experience. With sound-generating equipment rapidly becoming available, the tyranny of performance over composition is nearly at an end. Until now, in order to have your music played it had to be written in a more or less standard notation system which, as we shall see, has exasperating limitations. Added to this are the limiting capabilities of instruments and performers. Using a notation system you devise and program, a computer can control sound generation to the limits of its hardware capability under your direction.

If your musical experience is limited, you may be confused by my implication that methods outweigh notes in importance. Not much thought is needed to realize that the same pitches have been used and reused for a few centuries now. The composer's job, if something so enjoyable can be called that, is to dream up new ways to arrange those same old notes. Even if he decides to experiment with new tuning systems, in which the notes are not exactly the same as the old ones, the notes just "lie there" until he works out the rules for arranging them. A composition is a scheme in which musical sounds are ordered; composition is the act of designing such schemes. Whether the notes emerge from the composer's pencil or his computer is incidental to the total result that is the product of his mind.

Lowering old barriers does not insure that the newly accessible path is free of obstacles. So far, we have glossed over that part of the definition that says cybernetics is a comparative study. In cybernetic composition, you often will be painfully aware of the differences between human thought and computer processing. Besides facing technical problems related to computers, you may feel plagued by psychological ones based on the functioning of the human mind. Our senses evolved long

before we developed the ability to think abstractly and declare ourselves rational beings. Composing is one of the many human feats that requires knowledge as well as a good measure of those instinctual talents we employ subconsciously to make use of the information that comes to us through the senses. Not without difficulty, we can "explain" what we see, what we hear, and even what we compose. But far deeper and unruly is the task of describing *how* we do these things.

Analyzing and describing our own techniques of composition can be a traumatic exercise in self-psychoanalysis. And now, we must add to this the task of stating our description in a form that a computer can follow. Odd as it may seem, this ameliorates the entire problem, even making a solution possible in cases where otherwise it would not be. In order to describe "how" in human terms, we would begin with musical abstractions which, when you think about them, exist at a very high level of mental activity. We know very little about the lower levels at which these high-level thoughts are converted to electrical signals and channeled through the brain. So there is really no way to give a complete, accurate description of the human process.

On the other hand, any given computer has what is referred to as its *instruction set.* The instruction set contains every activity that can be performed at the lowest level of the computer's "neural anatomy." This is the level at which electronic signals manipulate the elements of data known as bits and bytes. The computer also has access to any number of high-level languages through which we can address that lowest level in a relatively painless way, analogous to the way we subconsciously command our own thought processes. In programming, we will convert high-level human ideas into a correspondingly high-level computer language. We can ignore the hidden steps that allow the computer to break down statements in this language into the detailed machinations of its instruction set.

Our sole concern is to describe a *model* of our methods, i.e., a program that begins with the same data that we use and ends by producing the same sort of results that we would produce. The trick in modeling is to proceed from start to finish with an objective description of a way to simulate one of your own techniques of composing. Although only a simulation, this forces you to collect your thoughts precisely enough to portray a complete representation of your technique. The subconscious elements in your style that refuse to show themselves must be bypassed by tangible procedures that are, for all practical purposes, equivalent to those refractory elements. Besides explaining your methods to the computer, you are explaining them to yourself in a way that does much to prevent creative ruts.

You can then alter your style easily by telling the computer to change a parameter here or a symbol there, but in your mind, each parameter and symbol will carry its own concrete musical meaning. Each represents a palpable control switch or adjustable dial in some element of your style. Creativity, stylistic control, and general musical thought can be practiced at mental levels unimaginable in the past.

2 Prelude to Programming

If any part of the problem faced by a composer in using a computer can be called fundamental, it is the broad one we will call *representation*. All the elements of his art and all his methods for manipulating them are potential subjects for "discussion" within a program. As such, they must be represented in a thoroughly self-consistent way. For example, if pitches are to be depicted as numbers, then operations such as transposition must transform these numbers into other numbers exactly as one would expect in terms of their pitch names. And in what follows, of course, that conditional "if" doesn't apply; pitches will indeed be represented by numbers.

MUSICAL PITCH

It is a fairly well-known fact that a given pitch has a vibrational frequency which can be expressed in terms of a number of vibrations per second (hertz). It is even more common knowledge that the standard piano keyboard has 88 keys. Though I certainly don't recommend it, if you were to remove a piano key you would very probably find a number stamped on one of its normally hidden surfaces. The lowest key on my spinet is indeed stamped with the value "1" and the highest with "88."

Because the keys and the well-tuned notes they generate lie in an absolutely fixed order, other sequential numbering schemes are possible, and some are even more convenient. For example, we could place zero somewhere around the center of the keyboard, say at middle C, and then use negative numbers for lower tones and positive numbers for higher ones. In dealing with melody in later chapters, we will use a convention that differs only slightly from this. Zero then will correspond to the pitch, C, that marks the beginning of some reference octave, not necessarily the middle octave.

In discussing scales or harmonic structures, even the notion of a reference octave may be ignored. The octave has a singular psychological effect when heard as a musical interval—and this fact has not been lost in the design of keyboard in-

struments. Anyone seeing a piano keyboard for the first time is struck by the repetitive pattern of black and white keys, in which the black ones appear in alternating groups of two and three. Mentally, we can extract one complete cycle of this pattern and number the dozen keys it contains.

We will choose the first note of the cycle to be the white one just to the left of the set of two black ones; this is a C-key and we will number it 0. The remaining white notes in the octave will then have the values 2, 4, 5, 7, 9, and 11. Their alphabetic names follow most unimaginatively from C to G, but then revert to A and B as though mocking our choice of a starting point. Of course, the initial note need not be C, but if it is, the complete correspondence between numbers and notes within the octave is the sequence shown in Fig. 2-1.

If we need to address octaves as well as notes, the most obvious approach would be, as described above, to extend our numbers as far as required to the left (with negative numbers) and to the right (with positive numbers) while simply repeating the pitch pattern over and over. The entire range of sound can be addressed this way and, if our choice of reference pitch proves inconvenient, we need only shift the numbers so that zero lies opposite whatever pitch is more suitable.

INTERVALS

Quite often, we will be more interested in the distance between notes than in the notes themselves. The distance between two notes is called a pitch *interval* and corresponds to the difference between their assigned numeric labels. At least, it does "chromatically," or in terms of the smallest unit of pitch separation used in our tuning system. Each such interval has been given a name that reflects the evolution of musical thought in terms of an alphabetic (or *diatonic*) pitch system. In the keyboard octave diagram in Fig. 2-1, the black notes have all been designated by their "sharp" nomenclature. In our 12-tone, "equal temperament" tuning system, D-sharp (D#) and E-flat (E♭) label the same pitch. Thus, depending on the spelling of the notes, an interval will have more

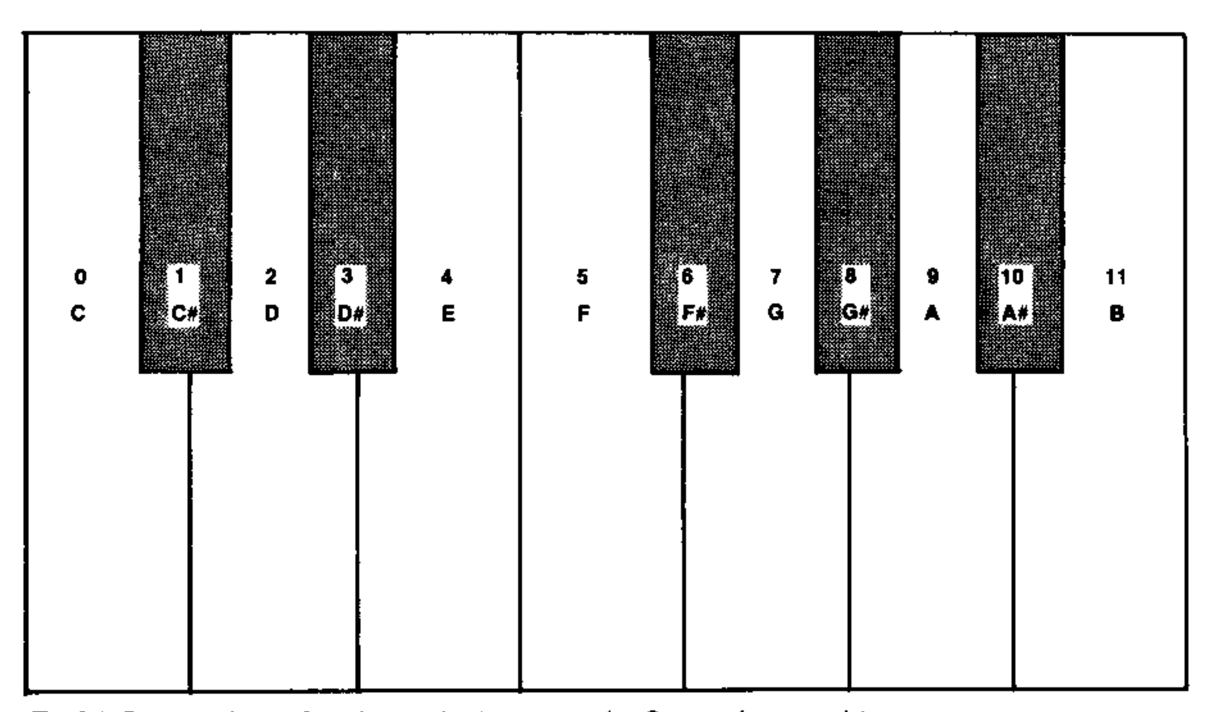


Fig. 2-1. Correspondence of numbers and note names, using C as a reference point.

	Interval	Names
0	perfect unison	(diminished second)
1	minor second	(augmented unison)
2	major second	(diminished third)
3	minor third	(augmented second)
4	major third	(diminished fourth)
5	perfect fourth	(augmented third)
6	augmented fourth or	diminished fifth
7	perfect fifth	(diminished sixth)
8	minor sixth	(augmented fifth)
9	major sixth	(diminished seventh)
10	minor seventh	(augmented sixth)
11	major seventh	(diminished octave)
12	perfect octave	(augmented seventh)

Fig. 2-2. Common musical names for pitch intervals from unison through octave.

than one name. The musical names assigned to the intervals (Fig. 2-2) may therefore seem confusing at first because they were devised in terms of "letter intervals" so that C to D-sharp cannot have the same name as C to E-flat.

Here, each of the common names is shown first with the less usual name following and parenthesized. Still more unusual names requiring modifiers, such as "doubly diminished," or ordinal numbers greater than eight (octave) have been omitted. Incidentally, two additional names are often used with regard to the intervals one and two when their spelling is uncertain or irrelevant. The first is called a *semitone*, *half tone*, or *half step*, and the second is a *whole tone* or *whole step*. Figure 2-2 is shown only as a translation aid. Personally, I would like to train myself to use only the numeric interval values, but too often I find myself using names in place of the numeric values, both in thought and speech, because of the musical sounds they induce in my mind through previous training. In "conversation" with a computer however, I will always use the numeric intervals.

The conventional idea of "key" in music normally would be introduced here. That idea, like the name of an interval, is based on the use of letters rather than numbers. Unlike the names of intervals, it is a topic of little importance in this study. As we shall see when we consider the concept of "tonality," the natural evolution of the notion of key has made the conventional idea obsolete.

CONSISTENCY IN REPRESENTATION

The operation musicians refer to as *transposition* can now be represented by the numeric operation of addition. It should be clear that in order to transpose all the notes in a given melody up a perfect fourth (check Fig. 2-2 if necessary), we need only add 5 to their numeric values. Similarly, to transpose down a minor sixth, we would subtract 8. However, suppose we are dealing not with melody, but with a musical component that is more conveniently thought of in terms of the pitch names or values in one reference octave. That is to say, we want all Cs to have the value 0 and all A-flats to be 8, regardless of their octave.

Focusing attention on a particular pitch, say E-flat or 3, we see that transposition up a perfect fourth can still be represented by adding 5 because,

$$5 + 3 = 8 \quad \text{or A-flat}$$

and this result lies in the reference octave. But suppose we were to carry out the operation "5 +" on 8, i.e., transpose A-flat up a perfect fourth. This gives the unlucky result 13, which is outside the range from 0 to 11 wherein lie our pitches. Composers face this problem continually; their more or

less intuitive method of solution must now be put into words, or else modeled so as to arrive at the same calculated result. An effective procedure would be to add the required number of semitones, divide the result by 12 (the total number of pitches), and retain only the remainder. Notice, the simple case $(5+3=8)$ conforms readily to this method because 8 (the result) divided by 12 has a remainder of 8. But the more recalcitrant $8+5=13$ also yields, because 13 divided by 12 has a remainder of 1, or D-flat.

If you think the main part of the results of division are being thrown away in favor of the dregs you are absolutely right. The number we have been discarding after division by 12 can be considered the octave designation. The value 13 really means C-sharp or D-flat *one* octave above the reference octave. (This italicized "one" is not the remainder, which also happens to be one, but the number of times 12 goes into 13.) Had we chosen for reference the octave in which the lowest pitch on the piano is represented by the value 9, then middle C, which is four octaves above our zero octave, would be represented by $0 + (4 \times 12)$, or 48. Notice that dividing this by 12 gives four (the octave) with a remainder of zero (the pitch). So the procedure described above doesn't really lose any information, it merely expresses it in a more convenient way—a way that is equally self-consistent and that allows us to discard information we don't need.

REPRESENTATION IN A PROGRAMMING LANGUAGE

In the discussion just given, we represented musical entities by numbers, but the entire discussion was presented in English so that definitions could be expressed in a mutually understandable manner. In talking to a computer, we are not concerned with conveying meanings; we simply want to tell it what to do. We must be able to refer to specific numbers as though they were things (such as pitches and intervals) and describe operations (like addition and the scheme for dividing by 12 using only the remainder) in a self-consistent way, so that the computer will produce results that can be translated back to us in a consistent and intelligible way.

The language we are going to use does not depend on the existence of any computer at all. This means it is a *programming language* rather than a *coding language*. It allows us to express our thoughts in an extremely concise way using letters, numbers, and some special symbols. The language is called by its unpunctuated initials, APL, rather than by the unassuming name for which they stand—A Programming Language[1].

Even when using a computer that doesn't speak APL, I program my thoughts first in APL and then translate that program into code acceptable to the computer. Fortunately, many computers can be addressed directly in this language. In describing the language, I will use notation that can be considered standard. However, in the implementation of APL on any given computer, the "standards" of notation may be somewhat different because few computers are designed to handle the set of characters needed by APL, either for input through a keyboard or for output to a display or printer.

A typical expression in APL would be:

```
NOTES ← 0 1 2 3 4 5 6 7 8 9 10 11
```

The left arrow may be read, ". . . is assigned the values(s) . . . ." So the above statement established an entity named **NOTES** which consists of a list of integers. Individual members of the list are easily addressed. You might expect **NOTES[1]** to contain 0, **NOTES[2]** to hold 1, and so on. However, there is a provision in APL that lets us count from 0 instead of 1 if we so desire. This is referred to as *index origin zero* and is put into effect by writing the statement:

```
□IO ← 0
```

From this point on, we will always assume our index origin or, in the jargon of APL, "quad IO," is

[1]K.E. Iverson, *A Programming Language*, (New York: John Wiley & Sons, 1962).

zero. As a result, we now have **NOTES[0]** equal to 0, **NOTES[1]**=1, . . ., and **NOTES[11]**=11. You may wonder why I sometimes say that a particular entry contains or holds a certain value, and at other times I equate it to that value. This is because names, such as NOTES[n], can be though of as labels similar to those on file drawers, and it is equally common to say, "This file contains A to K," or "This is the A-to-K file."

The statement in which **NOTES** was defined as a list of 12 consecutive integers required little effort to type on a computer keyboard. But suppose we want to define **PIANO** as a list of 88 entries beginning with 9 so as to simulate a standard piano keyboard. With far greater ease than in explaining, we can write

PIANO ← q + ι88.

In fact, the statement defining the shorter list could have been written

NOTES ←ι12.

The Greek symbol iota (ι) is used here as the *index generator* operator. The expression "ι12" is equivalent to the sequence of integers from 0 to 11. (If we were using index origin one, the same expression would be equivalent to the sequence from 1 to 12.) Assuming we have an APL computer, our entering just these three characters (not preceded by a name and left arrow) would cause the computer to respond by displaying

0 1 2 3 4 5 6 7 8 9 10 11 .

Similarly, typing "ι88" would produce a list from 0 to 87. In writing, "9+ι88," we add 9 to all elements of this list and so create the sequence

9 10 11 · · · 94 95 96 .

Preceding such expressions by a name and left arrow causes the computer to assign the results of the expression to the given name without actually displaying the results.

Like most APL operators, iota behaves differently when it is preceded as well as followed by values or names. It can be used to find the position of an entry in a list. For example, if we typed

NOTES ι 4

on our keyboard, the computer's response would be to display the index of the entry in **NOTES** containing the value 4. The result here would be 4 because the element with index 4 contains the value 4. But if we asked to see the "**PIANO** index of 4"

PIANO ι 4

the result "88" would appear, indicating that the value 4 is not contained in any entry because the highest index in the list is 87. But before going into further detail on the fundamentals of APL, we can now suggest ways to translate what was said about representation in English into APL.

To transpose E-flat up a perfect fourth, we can now write either

5 + NOTES[3] or **NOTES[3] + 5 .**

In fact, the entire octave could be transposed through the expression "**NOTES + 5**" to produce values from 5 to 16.

More usefully,

NOTES[0 3 7]

effectively spells out the elements of a three-note chord (a C-minor triad). To transpose this up a major third we could write

NOTES[0 3 7] + 4

which would give

4 7 11

the note values for an E-minor triad. Recalling how easily values can be transported outside of the reference octave, we can now introduce the APL *residue* operator whose symbol is a vertical bar. The expression

12 | NOTES + 5

would generate the sequence

5 6 7 8 9 10 11 0 1 2 3 4 .

Not only does this operation produce the remainders of **NOTES + 5** divided by 12, it also holds a convenient surprise.

In introducing the remainder algorithm, I purposely avoided mentioning the evasive actions that must be used to treat negative numbers. The APL residue operation takes such actions automatically so that

12 | NOTES − 5

produces the same sequence any competent composer would expect:

7 8 9 10 11 0 1 2 3 4 5 6 .

Notice the residue operation does not set aside our "octave designation" integer in a way that allows us to access it. But another and somewhat stranger symbol, ⌊, denotes the *floor* operation. It finds the largest integer that is not greater than its operand. That is, **⌊3.2** would be 3 and

⌊ 1.9 7.6 2.0

would be

1 7 2 .

So if we take the floor of 13 divided by 12

⌊ 13 ÷ 12

we do indeed retrieve the desired result, 1. Another helpful surprise takes care of negative numbers:

⌊ ¯1 ÷ 12

produces the integer ¯1, implying we have entered the octave just below that used for reference. In case you wonder why the result here is not zero, it is because zero is greater than a negative one-twelfth, and so cannot be the "floor."

One other question is bound to arise concerning the use of two different minus signs. In APL, the symbols for subtraction and negative value differ by their positions on the printed line: − versus ¯. As we shall see, this is necessary to distinguish between a symbol used as an operator and a symbol used as part of a number.

3 The Theme of APL

Much of the power inherent in APL lies in its ability to perform operations that treat lists of values as if they were single entities. In order to transpose the three entries of a minor triad, we needed only a single statement. It was equally simple to transpose all 12 notes in the octave, and we could just as easily transpose all the tones in a given piece! If we had to *code* such statements in another language, we would need to write programmed loops in which each entry in a list received individual treatment. There are more complex operations in which one list of values operates on another list. Coding such operations is extremely tedious and the many lines of required code offer ample opportunity to make mistakes. But in APL we can program even these operations as single statements.

VECTORS AND ARRAYS

We referred to the entities named NOTES and PIANO as lists of numbers. In APL, this particular form of list is called a *vector*. A vector is a list in which any particular entry can be addressed by a single index. We can also say that a *vector is a one-dimensional array*. Thus NOTES[3] refers to the fourth entry in that list—although to make this seemingly absurd connection between the index 3 and the fourth list entry, one may need to be reminded of our choice of index origin in which the index 0 comes first.

A two-dimensional list can be thought of in the form shown in Fig. 3-1. If the name assigned to this array in our program were LIST, the element marked by the X would be addressed as LIST[2;1]. The addressing scheme in APL requires the row index to precede the column index. The entire row of this array that contains the X would be the *vector*, LIST[2;1]. Similarly, the column containing the X is the vector, LIST[;1]. The role of the semicolon becomes particularly clear in the following.

LIST[2 3;]	means rows 2 and 3 (all columns)
LIST[2;4 1]	means the row 2 elements of columns 4 and 1

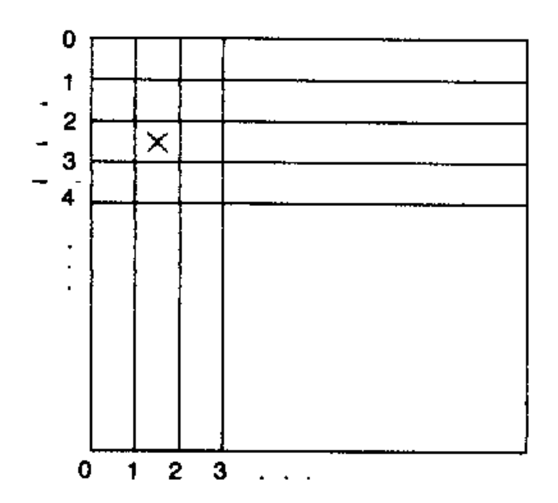


Fig. 3-1. Graphic representation of a two-dimensional array. Each position is addressed by the scale markings for its top and left side. Thus the address of the "X" is [2;1].

LIST[2 3;4 1] means the four elements means the whose row and column indices are [2;4], [2;1], [3;4] and [3;1], in that order.

Although a vector has only one dimension and is written in a horizontal format, we can say its "last" (and only) dimension is addressed by a column index. Now, on going from one to two dimensions, the old (column) index is preceded by the new (row) index. Similarly, to form a three-dimensional array we will place the new index first and think in terms of

[plane; row; column].

Figure 3-2 should help clarify the way the added plane index allows us to "collect" a group of two-dimensional arrays into one three-dimensional "file."

And now, four dimensions can be viewed with no mental gymnastics as sets of three-dimensional files or "file drawers." Again, the newest index (the drawer number) would be the first index given in addressing any particular entries. Sets of these, like file cabinets, would be five-dimensional objects. Although the process of adding dimensions can be pictured quite easily, it is rarely useful or even practical to go beyond three.

INDEXING

An element in an array can be indexed by any expression, as long as that expression, when evaluated, equals an integer that is a valid index for the array. For example, if

A ← 3 and B ← 5,

then NOTES[A + B] would be the same as NOTES[8]. However, NOTES[A × B] would be invalid because the index (15) would be out of range for this array of twelve elements. Further, NOTES[A ÷ B] violates the rule that an index must be an integer.

The entire array, NOTES, could be awkwardly written as shown below:

NOTES[0 1 2 · · · 10 11] or NOTES[⍳12]

and no rules of syntax would be broken. The point here is that any valid sequence of indices is accep-

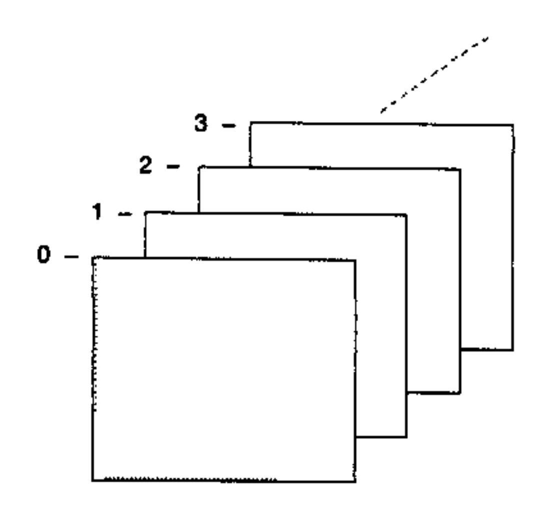


Fig. 3-2. A three-dimensional array addressed by [plane; row; column].

table. Thus if a "melodic vector," M, were defined as:

M←0 2 4 0 4 6 8 4

then **PIANO[48+M]** would pick out this melody in one-finger style beginning with middle C (48 on the keyboard). Moreover, if an array, A, had the form

0	4	7
2	6	9
4	7	11

then **PIANO[A+48]** would be valid and have the same shape as A! That is, even though **PIANO** is a vector, **PIANO[A+48]** is the three-by-three array:

48	52	55
50	54	57
52	55	59

The complexity of the APL expressions you use, as well as your depth of thought, will increase as you become fluent in the language. There are no bounds on complexity; as long as your statements are syntactically correct and do not generate more information than your computer can hold, you can say "anything" in APL. Thus, forms such as **A[B;C;D]**, where A is a three-dimensional array and B, C, and D are legitimate expressions, can be used to construct some very complicated multidimensional objects. I am not encouraging this, but merely pointing out that the power is there if you have the kind of thought processes that can make use of it.

THE SHAPE OPERATOR

The shape of an array is expressed as the numbers of elements in each of its dimensions. That is, the shape of **NOTES** (the vector of integers from 0 to 11) is 12. This is expressed in APL by writing

ρ NOTES

Typing that statement into our APL computer would cause it to fire back the value 12 on its output device. Entering instead the statement

S←ρ NOTES

would create a variable named S containing the value 12 within our program. If the operator ρ (Greek *rho*) were preceded by the numbers 3 4:

3 4 ρ NOTES

we would be specifying a new shape for the contents of **NOTES** instead of referring to the existing shape. A computer would respond by displaying the integers in three rows and four columns:

0	1	2	3
4	5	6	7
8	9	10	11

And of course, writing

T←3 4 ρ NOTES

would create the variable named T containing that array.

So the shape operator serves two important functions:

ρ A

captures the shape of A, while

B ρ C

reshapes the entries in C according to the specifications in the item B.

Suppose A is the array

1	1	2
1	2	1
2	1	1
1	0	0

in which each row gives the number of attacks on each beat in a bar of 3/4 time. That is, we could

be representing the rhythmic sequence shown on the top line of Fig. 3-3. The shape of A is 4 3 and it tells us directly that there are four bars with three beats in each bar. Setting

B ← 3 4 ρ A

establishes the array named B containing a representation of three bars with four beats in each derived directly from A. This reshaping operation proceeds by cycling over the existing indices in last-to-first order. That is, the consecutive sets of indices that label the consecutive entries in the array act just like the wheels on a mileage indicator except that, instead of each wheel having the digits from 0 to 9, each has the number of digits given by the entry in the same position in the vector to the left of the shape operator.

In this case, the "high-order" wheel contains the three digits 0, 1, and 2; the "low-order" wheel holds the four integers from 0 to 3. So the elements would be taken in the order:

[0;0], [0;1], [0;2], [1;0], [1;1], [1;2], [2;0], . . .

and the array **B** becomes

```
1 1 2 1
2 1 2 1
1 1 0 0
```

representing the rhythm shown on the middle line of Fig. 3-3. If we reshaped the array A into three rows and five columns:

3 5 ρ A

the indices would cycle just as on an odometer until the three rows (bars) of five elements (beats) are filled:

```
1 1 2 1 2

1 2 1 1 1

0 0 1 1 2
```

representing the bottom line of Fig. 3-3.

SCALARS AND EMPTY ARRAYS

Ironically, beginners find the idea of multidimensional arrays easier to grasp than the concepts of no-dimensional objects and empty arrays. An "array" with no dimensions is called a *scalar*. To give an analogy in simple geometric terms, the entries in a three-dimensional array in APL are addressed like the letters in a book, by:

[page; line; column].

A line of type corresponds to a vector in which each

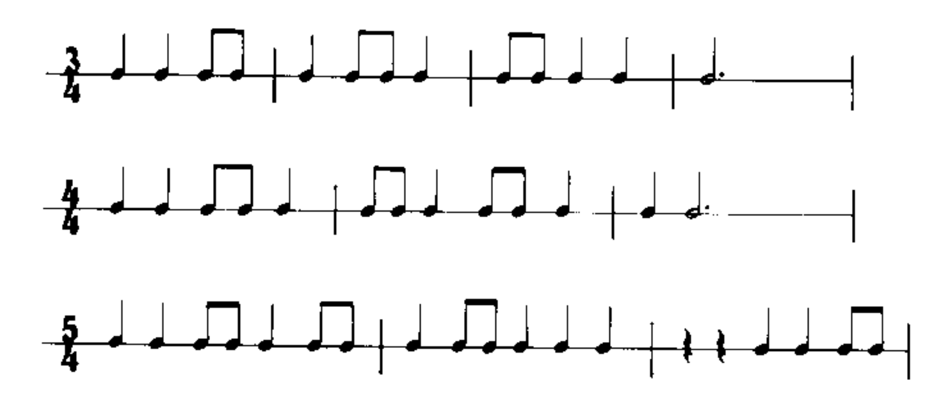


Fig. 3-3. The top line shows the rhythm of attacks represented by the array A, the second line is the attack rhythm represented by the expression B ← 3 4ρA, and the bottom line is a rhythm represented by 3 5ρA.

Object	Shape	Rank	Description
A	⍴A	⍴⍴A	
5		0	scalar
1 ⍴ 5	1	1	vector
1 1 ⍴ 5	1 1	2	matrix or 2-D array
1 1 1 ⍴ 5	1 1 1	3	3-D array
1 1 1 1 ⍴ 5	1 1 1 1	4	4-D array

Fig. 3-4. The shapes and ranks of some one-element objects, all of which contain the same single value.

letter can be addressed by one index. The line itself is reached through its page number and line index. Now, when we talk about a scalar, it is as though we're referring to a particular letter of the alphabet, say "h," without any indication as to where it appears in the book. The number 5 is a scalar; it has no "dimensions." However, it is also possible to have a vector or higher order array containing a single entry. Imagine a book with pages so narrow that each line can contain only a single character. Another book may be smaller still, having one character per page; and the ultimate in smallness would be a book containing a single character.

Now picture a book in which all the pages are totally blank. In terms of characters, this is an empty array. Note, an empty array may have some nonzero dimensions, but it has no elements. On the other hand, a scalar is one element with no dimension. The reason such oddities are required will become intuitively clearer with experience, but it can be summed up in the phrase, "self-consistent representation." And to complete the picture in a self-consistent way, the following statements sum up the properties of "shape" in APL.

1) The shape of any variable is a vector.
2) The shape of an empty array is a vector containing at least one zero.
3) The shape of an empty vector is the one-element vector, 0.
4) The shape of a scalar is the empty vector (a vector with no elements, not even a zero!)
5) The shape of the shape of a variable is a one-element vector giving the number of dimensions (the *rank*) of the variable.

Figure 3-4 shows the shapes and ranks of some one-element objects, all containing the same single value.

Syntactical Considerations

To create an array, the argument to the left of the shape operator must be a scalar or vector. If the array is to be empty, the left argument must contain a zero. The expression

```
0 ⍴ 1
```

creates an empty vector (0 elements equal to 1);

```
0 3 ⍴ 5
```

creates an empty two-dimensional array (0 rows with 3 columns). In these statements, the value to the right of the shape operator clearly doesn't matter; syntax simply requires the operator to have a right argument.

"The" empty vector can be written (⍳0 or iota zero) regardless of index origin. The expression

```
(⍳0) ⍴ A
```

creates a scalar (shape equal to the empty vector) containing the first element in A.

This level of detail becomes important when dealing with a computer directly in APL. In an ex-

pression such as A+B, the computer must be able to interpret exactly how to perform this operation. If it sees that A and B have entirely different shapes, it—and the programmer who wrote the program allowing it to "speak" APL—just doesn't know what to do. Many existing APL interpreters are not even clever enough to notice that single-valued objects such as those in the above table can be treated alike under certain conditions. Simply because their shapes are different, any attempt to combine them "improperly" will cause an error message to be displayed.

Notice that operators such as ⍴ consistently apply to whatever lies to their right. Parentheses must be used to delimit arguments. For example

```
⍴ A + B
```

is the shape of the sum of A and B, but

```
(⍴A) + B
```

adds the shape of A to B. The existence of a left argument can change the meaning of an operator significantly, and parentheses are often needed here, too. In fact, every valid APL expression has a rigid interpretation based on a right-to-left scan of its contents. Thus the expressions

```
 A + B ⍴ C  × D
 A +(B ⍴ C) × D
(A + B) ⍴ C  × D
(A + B ⍴ C) × D
((A + B) ⍴ C) × D
```

are all different.

As if the syntax of APL weren't hard enough to teach, a new version of the language has appeared (APL2), and its manual informs me that operators and functions are both "operations!" Following through with that misuse of English syntax, the author pleads with me not to call operators "operators." Unfortunately, I have already done so for years and will probably continue to until I can break the habit. According to the new cant, functions are operations that *may* have arrays as arguments. The name *operator* is reserved for operations that *must* have at least one function as an argument. There are many powerful, new features in APL2 which will not be covered here because they were not available when the programs to be described were written. The fact that the terminology makes no distinction between a surgeon and a tonsillectomy should not detract from its power as a programming language.

4 The Bass and Tenor of APL

No matter how proficient you become at programming and working with numerical representations, when you use your programs to compose, you will prefer to think in more musical terms. Because APL can handle letters as well as numbers, it is perfectly reasonable to write programs that accept literal information from the user, convert it to numerical representation, carry out the necessary numerical operations, and convert the results back into letters and names that are conveniently legible and meaningful. In APL, entries such as **PIANO** or **XXX** can be specified as names of variables containing information. Information is either numeric or literal. Names are differentiated from literal information through the use of single quote marks:

```
XXX ← ABC
```

versus

```
XXX ← 'ABC'
```

The first expression sets **XXX** to contain whatever information is associated with the name, **ABC**. The second makes **XXX** contain the alphabetic characters A, B, and C.

REPRESENTATION IN GENERAL TERMS

All data in APL are either numeric or literal. The first decision to be made in representing any concept within a program concerns the kind of symbols to use—words, names, letters, special characters, numbers. Various types of information could be defined as follows:

```
A ← 7.3
B ← 5
C ← 6.27 .01945 13 9.03
D ← 16 7 3 24 2
E ← 'X'
F ← 'F#7 ♭ 9'
G ← '↑ ↓ ⊖ < >  *'
H ← 1
J ← 0 1 0 0 1 1 0
```

These are all scalars or vectors, but such entries

are equally valid for arrays of higher order. The entries for A, B, C, D, H, and J are clearly numeric, but can be broken down further into three distinct types of numbers. In APL, all three types can be used in the same way as ordinary numbers, but each also has special properties and thus special uses not shared by the others. These must be taken into account whenever an appropriate representation is to be chosen for some musical entity.

The version of APL we're discussing will not permit a variable to be difined that is mixed numeric and literal!

```
K ← 7'B' 1.3 'R3'
```

will cause a *syntax error*. Literal variables may contain any symbols available on your hardware. In most of the musical examples to be shown, only the APL character set appears because such was the limitation built into my old printer. APL has a predefined *system variable* called the *atomic vector* (□ AV) which contains 256 characters, not all of them printable. Some of these are *control characters* such as carriage return, line feed, backspace, etc. But each has a specific position in the atomic vector and so can be addressed as □ AV[n] if necessary. Notice that

```
□ AV ⍳ 'A'
```

will return the index of the character "A" in the atomic vector.

Literals and Variable Names

The APL character set consists of uppercase letters, those same letters underscored, numbers, and quite a few special symbols. Although literals may contain any characters at all, even characters not in this set, the names of variables are more limited. A name may not contain any blanks and must begin with a letter or an underscored letter. Digits may appear in any but the first character. Some systems consider the underscore character itself as a legitimate "letter" and some even include the Greek delta (Δ) and underscored delta (⍙). You may use blanks freely in expressions to promote legibility. They are necessary only where numbers or names could be misinterpreted if not separated. To place a single quote in a literal, type a pair of single quotes:

'Ravel''s' will appear as Ravel's

and

```
'Ravel''s''''Pavane'''''
```

will print as

Ravel's "Pavane".

Instead of using the numerical representation given earlier for NOTES, a musician might prefer:

```
NOTES ← 12.2⍴'C C♯D E♭E F F♯G A♭A B♭B'
```

If the luxury of "♯" and "♭" symbols is not available on your system, the APL ↑ and ↓ characters will serve. Notice this is a matrix with 12 rows and 2 columns. A row index from 0 to 11 will select a pitch: NOTES[3;] contains the two characters "E" and "♭." A column index isolates the pitch letters or the accidentals. Also notice what would happen if we were not careful about using blanks in the definition of NOTES.

One might go so far as to define:

```
     C ← 'C'
CSHARP ← 'C♯'
     D ← 'D'
 EFLAT ← 'E♭'
```

etc.

If we now asked to see the shape of these variables, we would find that C is a scalar while CSHARP is a two-element vector! One apparent oddity here lies in the difference between the use of blanks in literals and numbers. A single number, no matter how many digits it contains, is a scalar. A single letter, which may be a blank character, is also a scalar. Two numbers, separated by a blank or a comma, comprise a vector. Two letters separated by a blank or a comma would be a *three*-element vector. And just to keep things consistent, there is

even a literal analog on the (numeric) empty vector. An entity expressed as two successive single quotes.

A ← ''

has all the properties of ⍳0 except for being literal rather than numeric.

There is one other difference between literals and numbers that can be troublesome. Suppose we were to define:

ONE ← '1'
TWO ← '2'
etc.

If we now asked to see **ONE** displayed, the digit "1" would appear without quotes. We might be aware that it is literal rather than numeric because of its position on the display—most systems indent for numbers and do not indent for characters. But we could easily overlook such clues and expect an expression such as **7 − ONE** to give a numeric result rather than an error message. Numeric operations cannot be applied to literals!

Numbers

For simplicity, consider just A, B, and H, which respectively were set to 7.3, 5, and 1 in the section on "General Representation." All three can be used with the APL operators +, −, ×, ÷ as well as with less common symbols which we will examine shortly.

The latter two numbers are clearly integers. In our kind of work, we need rarely be concerned about the way APL distinguishes between integers and nonintegers within a computer. But we must remember to use only integers when we index arrays. (In a melodic vector containing a string of pitch numbers, what would an index of 7.3—as in **MELODY[7.3]**—mean?)

Integers of the form in H and J above comprise a third class of numbers. Any numeric variable, no matter what its shape, that contains only ones and/or zeros can be used as an argument in a *logical* or *Boolean expression*. For the moment, you can consider the term "logical" to be applicable because 1 and 0 may represent yes and no or true and false. A more general understanding will emerge when we take up the relational and logical APL operators.

Input and Output

A commonly used term in computer jargon is I/O, which is pronounced "eye oh." This of course refers to input and output or the two-way communication between you and your computer. The forms of I/O we'll discuss here are the standard ones for APL. They support a standard input device (a keyboard) through which information enters the computer, and a standard output device (now usually a display) on which the computer prints whatever it has to say. (If any other devices are to be used, such as printers, disks, or other displays or terminals, your APL system will address them through system-dependent functions or through special programs called *auxiliary processors* or *device drivers*.)

It is probably evident by now that every expression you enter, even if it is only a number, will be "evaluated" and the result will have to go somewhere. We have seen that if the expression contains a left arrow, the result of that evaluation will be assigned to the name preceding the arrow. If no arrow is present, the result will be displayed on your output device. A right arrow (→) will use the result literally "to go somewhere," as we'll see when we discuss branching.

In entering expressions for direct evaluation, you can use APL as a calculator rather than as a programming tool. Of course, you can also ask for some very complex programs to be executed. In either case, this kind of input takes place in *execution mode*. In order to define any sequence of steps you want executed as a program, you must use *function-definition mode*. This is typical jargon for "a mode of operation in which the statements that comprise programs (called *functions*) will be accepted." For now, however, we are concerned with input and output during execution.

When executing, those predefined programs called functions can ask for input from the user.

Notice that the program will need to display some output if it is to ask for input. Within a program, a statement as simple as:

'ENTER INPUT'

can be used. Because there is no arrow, this literal is sent directly to the display (without the quotes).

The user's reply will enter the program through a symbolic window, a quad character (□) for numeric input or a "quad-note" (□) for literal input. The programmed statement to do this can be as simple as:

A □ .

But the window can also let information go directly into an expression that requires evaluation:

A ← B × □ + C.

Information also can leave the computer through the APL window. The statement:

□ ← A

is unnecessary because

A

alone will display the contents of A. But statements of the form:

□ ← A ← B × C

allow simultaneous assignment and display.

□ ← M ← ABC[1 2 0 1 5 5 3]

will store in M and display the indexed entries in ABC.

COMMON MATHEMATICAL OPERATIONS

The ordinary mathematical operators for addition, subtraction, multiplication, and so on are used in APL almost as you would expect. But there are two generalizations that apply which are unusual. First, all mathematical operators can be *monadic* or *dyadic*, that is, they may have one (right) argument or two (left and right) arguments. And second, all APL expressions are executed from right to left so that, in the absence of parentheses, APL operators treat everything to their right as one argument.

More explicitly, the plus sign in dyadic use represents ordinary addition.

T ← 5 + Y

will cause T to represent the sum of five plus whatever value Y represents. If Y is an array, the scalar 5 will be added to each element, and the resulting array, T, will have the same shape as Y. As a monadic operator, the plus sign is called the *identity operator* because it does nothing to change its argument (+A equals A).

The minus sign needs more careful consideration because of those two generalizations. First, the need to distinguish between the monadic *negation operator* and a negative number requires the introduction of an additional symbol which is not an operator. As pointed out earlier, the numeric minus sign is a short dash printed noticeably higher than the usual symbol. The expression

¯5 ¯17 ¯4

is a vector containing three negative integers. This sign can only be used in this way, that is, as part of a number! The monadic negation operator (−) negates the sign of everything that follows (in its right argument). The expression

−K− ¯17

says, "Subtract negative 17 from the value of K and change the sign of the result."

−4+ ¯17 would be 13
−21+ ¯17 would be ¯4.

As a dyadic operator, it functions quite normally:

13 − 5 is indeed 8.

But notice:

−4 − ¯17 would be ¯21

while

− ¯4 − ¯17 would be ¯13.

The dyadic multiplication and division operators behave as you would expect:

5 × 2 equals 10 and
5 ÷ 2 equals 2.5

But as monadic operators:

× A implies the *signum* of A
÷ A represents the *reciprocal* of A.

For those unfamiliar with these terms, the signum is 1, 0, or ¯1 if the argument is greater than, equal to, or less than zero, respectively. The reciprocal of an expression is one divided by the expression.

× 5 × 2 equals 1 (the signum of 10)
÷ 5 × 2 equals .1 (the reciprocal of 10).

Some may also need to be reminded that a logical definition of the process of division shows that as the denominator decreases, the result becomes larger. In fact, as the denominator approaches zero, the result exceeds any definable number. Therefore division by zero is undefined, and any expression in which the right argument of the division or reciprocal operator is zero will cause a *domain error*.

In general, very small and very large values can be troublesome when working with a computer simply because of restrictions built into the hardware. The manual for each implementation of APL will state the maximum positive and negative values that can be handled. Numbers close to zero (not integers) are treated in a way that depends on the value of another system variable called *comparison tolerance* (⎕ CT) or, more colorfully, "fuzz." Usually this has a default value of about 10^{-13} (one over 10 trillion), which is too small to account for certain other problems that computers have in performing arithmetic with small numbers. We'll return to this later.

RELATIONAL OPERATORS

Before considering the other mathematical operators, we'll look at those that express a relation between two values. By definition, "between two values" implies these must be dyadic operators, and the "relation" between the two arguments will evaluate to either one or zero, indicating whether it is true or false.

Equal (=) and not equal (≠) symbols can be used in both numerical and literal expressions to produce logical (1 or 0) results. Don't confuse the equal sign with the assignment ← operator.

K = 5

gives the result 1 only if K does, in fact, equal 5. If K were any other value, the result would be 0. Notice, if K were the vector 3 5 7 9 7 5 3, the result would be the logical vector, 0 1 0 0 0 1 0.

B[3] = 'G'

produces a result of 1 only if the third element of the vector B is the character "G." If S were set to the literal vector 'POOR', then S = 'GOOD' would be the logical vector 0 1 1 0. The expressions

K ≠ 5 and
B[3] ≠ 'G'

would be true (equal to 1) only if the not equal relations were true. S ≠ 'GOOD' would equal 1 0 0 1, if S were actually equal to 'POOR'.

The remaining relational operators apply only to numerical arguments:

A	< B	equals 1 if A is less than B,
A	≤ B	equals 1 if A is less than or equal to B,

A > B equals 1 if A is greater than B, and

A ≥ B equals 1 if A is greater than or equal to B.

Setting

A ← 1 2 3 4 5 and

B ← 6 5 4 3 2, causes

A < B to equal 1 1 1 0 0 and

A ≥ B − 1 to be 0 0 1 1 1.

LOGICAL OPERATIONS

There are five logical operators formed from the three symbols ~, ∧, and ∨.

The monadic NOT operator (~) changes the value of a logical expression from 1 to 0 or 0 to 1. It can also be used in conjunction with the other two operators, both of which are dyadic, to "reverse" their meaning.

The AND operator (∧) gives a true (1) result only if both of its arguments are true. The expression:

(A = B) ∧ (C > D × E)

equals 1 only if A equals B and C is greater than the product of D and E.

Combining NOT with AND produces the NAND operator (⍲). As the phrase "NOT AND" suggests, it gives results exactly opposite to those produced by AND; only if both arguments are true (1) is the result false (0).

The remaining symbol (∨) is the OR operator. A ∨ B is true (i.e., 1) if either (or both) arguments are true. An OR relation is false (0) only if both arguments are false.

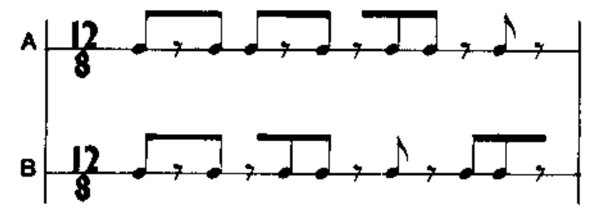


Fig. 4-1. Two rhythms to be represented by the logical vectors A and B.

The NOT operator combines with OR for the NOR operation which negates the simple OR. It is true (1) only if both its arguments are false (0).

As an example of utility, let the value 1 (true) represent a rhythmic attack and the value 0 (false) indicate no attack. To account for the three beats in a bar of waltz time, we might write the rhythm for a bass part as:

BASS ← 1 0 0

For an "oom-pah-pah" effect, we can now let the trumpets play the rhythm:

TPTS ← ~BASS

which would be 0 1 1. For an example with more interest, consider the two rhythms shown in Fig. 4-1, which can be represented logically as:

A ← 1 0 1 1 0 1 0 1 1 0 1 0

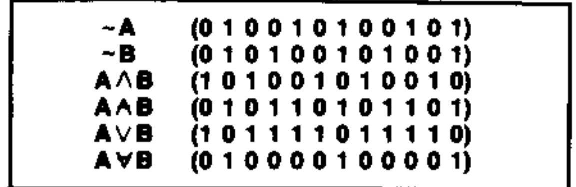
and

B ← 1 0 1 0 1 1 0 1 0 1 1 0.

Using the relationships depicted in Fig. 4-2, we now can produce the rhythms shown in Fig. 4-3. It is not unusual for the direct use of simple logic to produce such full musical settings. The same logic can also represent all sorts of schemes for ordering these parts sequentially instead of playing them all simultaneously. But we'll get into that later.

CONFORMABILITY

For the two arguments in a dyadic operation



~A	(0 1 0 0 1 0 1 0 0 1 0 1)
~B	(0 1 0 1 0 0 1 0 1 0 0 1)
A∧B	(1 0 1 0 0 1 0 1 0 0 1 0)
A⍲B	(0 1 0 1 1 0 1 0 1 1 0 1)
A∨B	(1 0 1 1 1 1 0 1 1 1 1 0)
A⍱B	(0 1 0 0 0 0 1 0 0 0 0 1)

Fig. 4-2. Relationships created by performing the NOT, AND, NAND, OR, and NOR logic operations on vectors A and B.

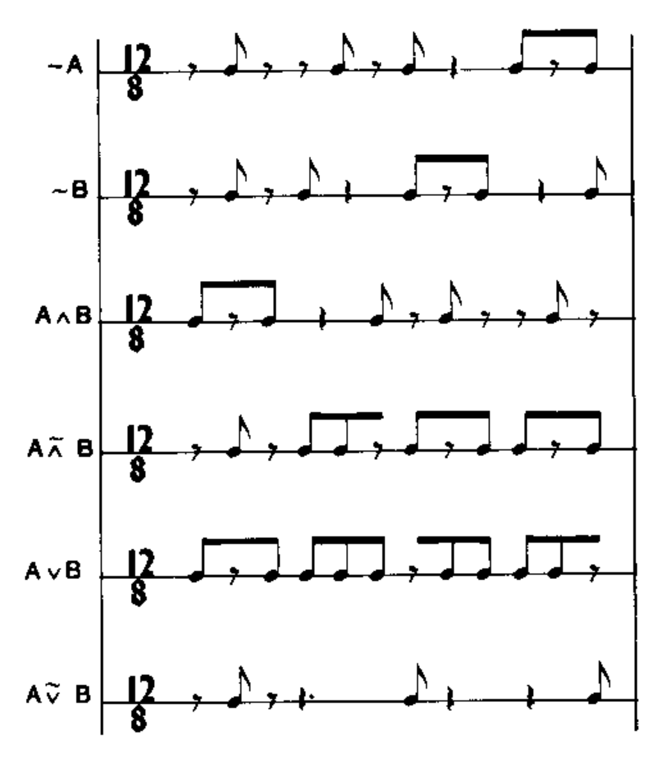


Fig. 4-3. Rhythms derived from vectors A and B through "logical operations."

to be compatible partners, their shapes must be "conformable." Usually, this means they must either have the same shape or else one of them can be a scalar. In general, there are different conformability requirements for different operators. Where the usual conditions just given do not hold, the restrictions on the shapes of the arguments will be given explicitly. Notice, if

A←2 2⍴1 2 3 4,

then A is

1 2
3 4

⍴A is

2 2

1 + A equals

2 3
4 5

A × A equals

1 4
9 16

A×A×A equals

1 8
27 64

and

1 0
¯1 ¯2

is the result of A−A × A−1.

5 The Instruments of APL

Of the remaining APL operators, some will appear stranger than others, but more importantly, some will be more useful than others. Their utility will depend on your own way of doing things and your understanding of their pertinent mathematical and logical properties. As your programming techniques increase in sophistication, you will find surprising uses for many of the operators. However, some APL operators, though very powerful, will be omitted here because they are not directly applicable to the subject at hand. For anyone interested in "composing" his own instrumental sounds, this can be a grave omission.

EXPONENTIALS AND LOGARITHMS

The asterisk (*) in dyadic form raises its left argument to the power(s) designated by its right argument. **A*3** means that every element of A is to be raised to the third power (A×A×A). Of course, the exponent (3, here) does not have to be an integer or a scalar. The fundamental mathematical properties you need to understand in order to use exponents intelligently can be found in any algebra text at the first-year college level.

The most obvious uses of exponentiation lie on the mathematical side of the boundary between composing and programming. Exponentials are indispensable in calculating pitch frequencies in any "equal-tempered" tuning system. They can be extremely useful in shaping waveforms. But these are topics we cannot cover here. One of their most useful properties for our purposes lies in the "tricks" they play with the number minus one:

(¯1)*(any odd integer) equals minus one.
(¯1)*(any even integer) equals plus one.

While I recommend looking into their properties if you are not familiar with them, I can assure you that if you ignore their existence, you'll experience very few ill effects.

As a monadic operation, the asterisk will be of little use to us here. It simplifies use of the transcendental number, *e* (the base of natural

logarithms), in expressions. That is, instead of writing 2.7182818, one could use *1.

The inverse function to exponentiation is symbolized by an encircled asterisk (*). If A raised to the B power is C (A*B equals C) then the logarithm of C to the base A is B (A * C equals B). The inverse nature of these two operators carries over to their monadic forms. Thus, * A is the natural logarithm of A (the logarithm of A to the base *e*).

ABSOLUTE VALUE AND RESIDUE

A much simpler and more generally useful symbol is the vertical bar introduced earlier. The "absolute value" of A (|A) refers to the magnitude (the numeric value without regard for the sign) of A:

| 3 ¯7 .5 ¯1.7 equals 3 7 .5 1.7

We have already used the dyadic form of this operator. Remember, A | B (or the A *residue of* B) is the remainder found on dividing B by A with "suitable adjustments" mad for minus signs:

3 | ¯3 ¯2 ¯1 0 1 2 3 equals 0 1 2 0 1 2 0

but

¯3 | ¯3 ¯2 ¯1 0 1 2 3
equals 0 ¯2 ¯1 0 ¯2 ¯1 0.

MAXIMUM, MINIMUM, FLOOR AND CEILING

A pair of operators identical in concept but exactly opposite in functions are ⌈ and ⌊. In monadic form, these are the *ceiling* and *floor* operators. The "ceiling of A" (⌈A) produces the smallest integer that is not less than A and the "floor of A" (⌊) implies the greatest integer that does not exceed A. The floor of 7.5 (⌊7.5) is 7 while the ceiling (⌈7.5) is 8.

⌊ 1 1.0 1.1 1.9 2 equals 1 1 1 1 2
⌊ 1 1.0 1.1 1.9 2 equals 1 1 2 2 2

Dyadically, these operators select the maximum or minimum of their arguments:

Expression	*Result*
7.906 ⌈8.01	8.01
8 ¯3 2 ⌊¯6 ¯1 0	¯6 ¯3 0
0 ⌈1.3 ¯9 17	1.3 0 17

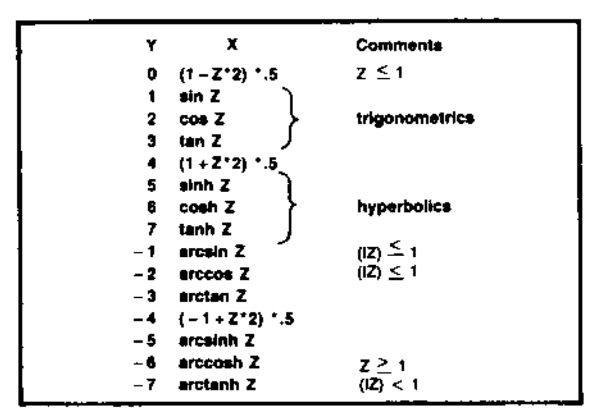
CIRCLE FUNCTIONS

The name and form of this operator (○) were suitably chosen to suggest its functions which relate to various properties of circular geometry. In its simplest form, ○ 1 is equivalent to π (3.14159 . . .). More generally, "circle A" (○ A) results in π times A.

If a left argument is used, it must be an integer in the range from − 7 to 7. There are constraints on the right argument that depend on the value of the left argument. Even if you are well acquainted with the trigonometric, hyperbolic, and Pythagorean functions represented by these 15 dyadic forms, you will find little or no use for them in what follows. (They can be essential however if you want to synthesize your own waveforms!) My main reason for mentioning them here is to point out that a variable—here, the left argument—can represent something quite abstract and yet be ordered in a scale (¯7, ¯6, ¯5, . . .). Figure 5-1 shows the meaning and restrictions on the expression X ← Y Z.

PERMUTATIONS AND COMBINATIONS

The exclamation mark is used to represent the factorial function. The factorial is a product of con-



Y	X	Comments
0	(1 − Z*2) *.5	Z ≤ 1
1	sin Z	trigonometrics
2	cos Z	
3	tan Z	
4	(1 + Z*2) *.5	
5	sinh Z	hyperbolics
6	cosh Z	
7	tanh Z	
− 1	arcsin Z	(IZ) ≤ 1
− 2	arccos Z	(IZ) ≤ 1
− 3	arctan Z	
− 4	(− 1 + Z*2) *.5	
− 5	arcsinh Z	
− 6	arccosh Z	Z ≥ 1
− 7	arctanh Z	(IZ) < 1

Fig. 5-1. Meaning of and restrictions upon the circle function as used in the expression X ← Y○Z.

secutive integers from one to the argument of the function. Thus, "seven factorial" (!7) equals 1 × 2 × 3 × 4 × 5 × 6 × 7, or 5040. Using an inductive approach, this function can be generalized so that a value can be calculated when the argument is any number at all except a negative integer. That is to say, !N has meaning (and can be computed) when N has any positive or negative noninteger value as well as when N is an integer greater than or equal to zero (!0 equals 1).

When N is an integer, the result of !N has a simple and, for us, useful interpretation. It equals the number of ways that N different things can be arranged in sequence, i.e., the number of permutations of N items. Three factorial (!3) equals 1 × 2 × 3 or 6; given the three items a, b, and c, there are six ways to order them.

To display a set of permutations, you need some sort of scheme that will let you keep track of the items to ensure that each permutation appears just once and individual items in one permutation are not replicated. The most useful scheme in working with computers is purely numeric:

1)	1 2 3	a b c
2)	1 3 2	a c b
3)	2 1 3	b a c
4)	2 3 1	b c a
5)	3 1 2	c a b
6)	3 2 1	c b a

Each row listed here shows the index for the permutation (from 1 to 6), the corresponding permutation for the integers from 1 to N (here, N = 3), and the same permutation for the items themselves. The idea here is to think of the permutations of the integers as N-digit numbers and arrange them in increasing order. Notice 123 is less than 132 which, in turn, is less than 213, and so on. Then using the integers as indices of the things to be permuted, we can write "a" wherever "1" appears, "b" in place of "2," etc.

The dyadic operation, M!N, represents the number of ways that M objects can be selected from a set of N objects. For the four things, a, b, c, and d, we find

1!4 equals
2!4 equals
3!4 equals
4!4 equals
5!4 equals 0

showing there are four ways to extract a single item (a, b, c, or d), six ways to choose two items (ab, ac, ad, bc, bd, or cd), four combinations of three things (abc, abd, acd, or bcd), one way to pick all four, and zero ways to take five from a collection of only four.

We will see the usefulness of these operations in thematic development and in the choice of raw material for composition.

NUMBER SYSTEMS

Two more mathematical operators are used only in dyadic form and deal with a fundamental set of concepts needed to understand much about computers and computer jargon.

The *decode* operation (⊥) can best be grasped intuitively using the value 10 as its left argument:

```
10 ⊥ 4 7 2 5
```

yields the value 4,725. The right argument (a vector here) is "decoded" to a value in a number system defined by the left argument. With 10 as the left argument, we're decoding the vector to an ordinary decimal or "base-10" number.

To understand that this process is not just a matter of removing the spaces between the digits in the vector, let's model the operation by using some simpler APL ideas. Consider the three vectors:

```
A ← 4 7 2 5
B ← 3 - ⍳ 4
C ← 10 * B
```

Here we see the original vector defined as A. Next, you should examine the vector B to verify that it equals 3 2 1 0. Similarly, C will be found to contain 1000 100 10 1. Now by multiplying A × C (or A × 10 * B), we find 4000 700 20 5 which, when added together, give the final result, 4725.

We can apply the same process to:

```
8 ⊥ 4 7 2 5
```

by using

```
C ← 8*B
```

and find a result of 2,517. In like manner:

```
16 ⊥ 4 7 2 5
```

will be found to to yield the value 18,213.

The subject of number systems can seem rather sophisticated to beginners, but much of the mystery in computer terminology evaporates after being exposed to this subject. So let's back off briefly to reconsider in a more elementary way the ideas just introduced. The numbers we ordinarily deal with contain the digits from 0 through 9. The two-digit number, 10, represents the base of this number system, and gives it the name, *decimal*. In fact, a one followed by N zeroes represents the base raised to the power N. For a number in the *nonal* or base-9 number system, digits from 0 to 8 would appear and the *number* 10 would have 9 as its *value*! In this way we can device number systems with any base down to 2, and express numbers in such systems in terms of their (decimal) *values*.

In the base-2 or *binary* system, the digits are restricted to the range from 0 to 1 and the number 10 must have a value of 2. The term *bit* is a contraction of "binary digit." The fact that electronic signals can be classified in terms of two possible states—above or below some threshold—allows circuits to manipulate bits in perfectly logical ways so as to mimic mathematical algorithms. Computers manipulate bits in groups rather than singly, and the size of these groups determines the size of the numbers that can be dealt with. Just as N decimal digits can represent 10*N different values, N binary digits can run from zero to (2*N) −1.

In working with high-level languages such as APL, one can remain isolated from these troublesome details. But somebody must be able to work with the computer on its own terms in order to design and implement such languages and to determine the source of errors when something goes wrong at a fundamental level. Humans have as much trouble recognizing long strings of bits as they do reading long sequences or musical notes without bar lines of ligatures.

But patterns of bits as well as of sixteenth notes are quite easily grasped by the mind when grouped in threes or fours. An earlier generation of computers grouped bits into *words* whose length was divisible by three. Octal numbers were then convenient to use because each octal digit is equivalent numerically to one of the eight possible 3-bit patterns from 000 to 111. On going to a word length divisible by four, the base-16 or *hexadecimal* system came into common use. Notice that this system needs 16 unique digits to represent the values from 0 to 15. The representation that has become standard uses 0 to 9 followed by the letters A through F, so that the first two-digit number (10) has the required value, 16. A *byte* is defined as eight bits or two "hex" digits, and is easily pictured by musicians as a bar of eighth notes and/or eighth rests in 4/4 time.

Figure 5-2 shows the equivalent representations for the values from 0 to 16 in the four most-used number systems.

Many kinds of measurements, such as time, angle, or distance, require systems for counting that are not restricted to a single base. The expression

Decimal	Octal	Hexadecimal	Binary
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	10	8	1000
9	11	9	1001
10	12	A	1010
11	13	B	1011
12	14	C	1100
13	15	D	1101
14	16	E	1110
15	17	F	1111
16	20	10	10000

Fig. 5-2. Representations of the first 17 integers in decimal, octal, hexadecimal, and binary notation.

```
1 5280 12 16 ⊥ 17 650 9 13
```

uses on the left the conversion factors for feet per mile, inches per foot, and the number of sixteenth-inch scale markings per inch. The right argument shows a measurement of 17 miles, 650 feet, 9 and 13/16 inches. The result of this expression is 17,358,877 which is the number of sixteenths of an inch in this distance! In the same way,

```
1 8 4 12 ⊥ 3 6 2 9
```

would tell us the number of 1/32nd note durations that are needed to reach the ninth 32nd note of the second beat in the sixth bar of the third phrase, where there are twelve 32nd notes per beat, four beats per bar, and eight bars in a phrase.

To understand the algorithm used here, let's return to the base-10 example where the vector A was set to 4 7 2 5. Another way to get the same answer in that case would have been to define:

```
B ← 10 10 10 1
C ← 10 10  1 1
D ← 10  1  1 1
```

and then set the result to the sum of the elements in A×B×C×D. Using this same pattern for the left argument in the above example to find the number of 32nd notes:

```
A ← 3 6 2 9
B ← 8 4 12 1
C ← 4 12 1 1
D ← 12 1 1 1
```

Now, summing over A×B×C×D gives

$$(3\times 8\times 4\times 12) + (6\times 4\times 12) + (2\times 12) + 9 = 1473$$

There are additional possibilities for the arguments here. The only one I will point out is the case where the right argument is a matrix. Say A is the array:

```
1 2 3
4 5 6
7 8 9
```

Now, 10 ⊥ A will give the vector 147 258 369.

Under certain conditions, the *encode* function (⊤) is the inverse of decode (⊥). The qualification is needed because the shape of the left argument has a critical effect on the result. For example:

```
            10 ⊥ 4725   equals 5
         10 10 ⊤ 4725   equals 2 5
      10 10 10 ⊤ 4725   equals 7 2 5
   10 10 10 10 ⊤ 4725   equals 4 7 2 5
10 10 10 10 10 ⊤ 4725   equals 0 4 7 2 5
```

We can write an expression using only previously introduced operations to produce the same result:

```
10 | ⌊ 4725 ÷ 1000 100 10 1
```

Scanning from right to left, we see the first operation divides 4725 by the vector 100 100 10 1, producing 4.725 47.25 472.5 4725. The next operator to the left is the floor. It can't be the dyadic minimum because there is another operator to its left! Executing the floor operation gives 4 47 472 4725. Finally, we take the 10 residue of this to arrive at the expected result, 4 7 2 5.

You might try using this method, replacing the powers of 10 with suitable values, to show that

```
8 8 8 8 ⊤ 2517
```

and

```
16 16 16 16 ⊤ 18213
```

both return the original vector, 4 7 2 5. You might now see if you can alter the algorithm to show

```
1 8 4 12 ⊤ 1473
```

returns the vector 3 6 2 9.

The extension to more complex arguments will

only be suggested here by stating that

10 10 10 ⊤ 147 258 369

will return the matrix

```
1 2 3
4 5 6
7 8 9
```

RAVEL

You are probably aware of an intrinsic difference between the "mathematical" operators we've been discussing and the shape operator, ⍴, manipulate variables in ways that often show no regard for the *values* of these variables. The first such operator uses the comma as its symbol. With only a right argument,

,A

is called the *ravel* of A and produces a vector containing all the elements of A in "mileage-indicator" order, as described in the section on the shape operator. If A were a scalar, its ravel would be a vector with a single element. The ravel of the matrix

```
1 2 3
4 5 6
7 8 9
```

is the vector 1 2 3 4 5 6 7 8 9.

CATENATE

When used dyadically, the comma serves its usual linguistic purposes of chaining, or *catenating*, things together. The catenation of A and b would be written as **A,B**. For the moment, consider the ranks of both arguments to be less than two, i.e., is a scalar or vector. Their catenation is the vector containing both sets of elements. Notice that number can be strung together with spaces or commas, so that

1,2,3,4,5 equals 1 2 3 4 5.

But the comma must be used to catenate numeric variables with numeric constants (ordinary numbers) or with other numeric variables. The expression

A,1 2 3,B

equals the elements of A followed by 1 2 3 followed by the entries in B. If A were the vector 1 2 3, then
A,A,A
would be 1 2 3 1 2 3 1 2 3.

Frequently we will have reason to use a statement in which the empty vector is catenated to something. From a purely practical viewpoint, this may appear senseless, but we will see that there are good reasons for doing so when we consider the logic involved.

Character (literal) scalars or vectors may also be catenated (but not with numerics):

```
A ← 'MAJ'
B ← 'MIN'
C ← 'OR '
```

so that

A,C,C,B,C

equals

'MAJOR OR MINOR'

(Note that C[2] equals a blank character.)

CATENATION WITH THE "AXIS OPERATOR"

Arrays with rank greater than one can be catenated using the form

A,[N]B

where N is an integer. A bracketed value appearing after an operator in this manner is often referred to as the *axis operator*. However, for this operation

to be acceptable, the shapes of both arguments must meet certain conformability requirements. If you were given objects shaped like A and B at the top of Fig. 5-3, and were asked to join them together so as to form another well-defined array, you would undoubtedly derive either the "A,B" or "B,A" configuration shown in that figure, rather than irregular shapes such as those marked "invalid." And, of course, you would be correct indoing so.

Now notice that these are two-dimensional or rank-2 objects and, from the way they are drawn, the shape of the newly formed object has the same first (row) entry as do the shapes of A and B. The second component of the shape, or the column entry, distinguishes the new array. The bracketed N for this operation would equal the index of the altered dimension in the shape. Here that would be the second (last) component—remember, the order for matrices is row,column—so we would write A,[1]B, remembering also the effects of index origin zero on labelling the components of any vector. Actually, when N refers to the last (column) dimension, the axis operator may be omitted. Thus A,B will have the same effect as A,[1]B here. If A and B were rank-3 objects, then A,B would give the same result as A,[2]B.

In general terms, for two arrays to be conformable for catenation along the [N]-axis, all the other components of their shapes must match. Notice that this implies the two arrays can also have ranks that differ by one if the unmatched axis [N], is the "missing" one in the array of lower rank.

A most convenient feature allows a scalar to be catenated to an array by expanding the scalar along the required axis. For example, let A be the

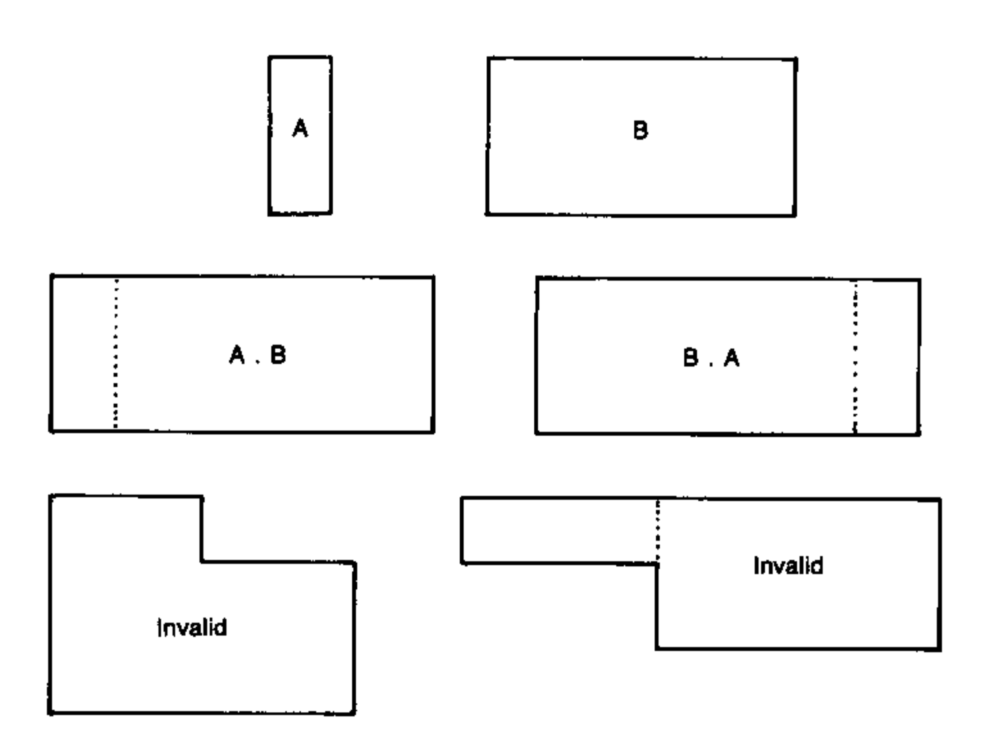


Fig. 5-3. A and B are arrays having the same number of rows. They may be catenated in either of two ways, forming a larger array that also has the same number of rows. Joining the two in any other way will not produce a "well-defined" array.

array 2 3 ⍴ 'ABCDEF' and B the scalar (single character), '*'. Now A,B is

```
ABC*
DEF*
```

B,[0]A is

```
***
ABC
DEF
```

B,A,B equals

```
*ABC*
*DEF*
```

And B, [0] (B,A,B), [0]B produces

```
****
*ABC*
*DEF*
****
```

LAMINATION

If two arrays have identical shapes, they can be joined as adjacent members of a larger array using the *laminate* operation. Thus two vectors of length L can be laminated to form a matrix with two rows and L columns or L rows and two columns. To do this, the comma and axis operators are used as before except that, in the case of lamination, the bracketed N is *not* an integer. The value used for N indicates where the new axis is to be inserted into the vector of existing shape elements. In the case of two vectors is length L,

```
A,[¯.5]B
```

would create two rows, while

```
A,[.5]B
```

would create two columns. The following table generalizes the situation for two arrays with shapes equals to x, y, z, ...

A,[¯.5]B sets shape to 2, x, y, z, . . .
A,[.5]B sets shape to x, 2, y, z, . . .
A,[1.5]B sets shape to x, y, 2, z, . . .
A,[2.5]B sets shape to x, y, z, 2, . . .

Notice that N is chosen to lie between the integers that index the current shape vector or within the bounds of the next integer at either extreme. This means the fractional part of N is not restricted to the digit 5 and, if the index origin were one, the N value would still obey the same rule.

Lamination can also be carried out if one argument is a scalar. APL simply extends the scalar to match the shape of the other argument. Setting

```
A ← 2 3 ⍴ 1+⍳6.
```

A,[¯.5]0 would form:

```
1 2 3
4 5 6

0 0 0
0 0 0
```

where the new shape is 2 2 3. Writing instead:

```
A,[.5]0
```

would create another 2×2×3 array, but the second (row) axis is now the new one:

```
1 2 3
0 0 0

4 5 6
0 0 0
```

MEMBERSHIP

The Greek epsilon, ∊, is used only in dyadic format and represents the *membership* operator. Comparison with the dyadic iota ("index of" operation) is instructive and also helpful in avoiding confusion between them. Consider these two expressions:

```
I←V ⍳ A
M ←A ∊ B
```

The variable A represents the same object in both expressions and can have any shape. We can now clarify certain aspects of the first expression which were avoided earlier because the required concepts had not been established.

The shape of I will equal the shape of A and, for each element in A, the corresponding element in I gives the first index in V that contains that A element. V must be a vector.

In the second expression, the shape of M will also equal the shape of A, but notice that A is the *left* argument of ϵ. There are no conformability requirements for the arguments; A and B may have any shapes. The result, M, will be a *logical* array that tells whether each entry in a is present anywhere in B. The expression AϵB can be read, "A is a member of B." Notice, ~AϵB represents the statement, "A is not a member of B."

If we let V equal 1 2 3 4 5 6, and set W to the array:

```
1 3
3 5
1 7
```

the following relations hold:

```
V ϵ W equals     0 2
                 2 4
                 0 6

V ϵ W equals     1 0 1 0 1 0

W ϵ V equals     1 1
                 1 1
                 1 0
```

TAKE AND DROP

A pair of operators that are useful in selecting only parts of arrays are represented by up and down arrows; these are both dyadic. Their left argument can be a scalar or a vector with as many entries as are in the shape of the right argument. As a base for examples here, we'll set:

```
A ← 'ABCDEFG'
B ← 1 + ι 7
C ← 4 5 ρ 'RED  GREENBLUE BLACK'
D ← 3 6 ρ ι18
```

Now, the expression ↑ A will select the first four elements of that vector, ABCD; the expression ¯4 ↑ A selects the last four elements, DEFG. If the left argument is larger than the number of elements in the right argument, blanks of zeroes will be appended to the result according to whether the right argument is literal or numeric:

```
 9↑ A => 'ABCDEFG  '
¯9↑ B => 0 0 1 2 3 4 5 6 7
```

For the higher order arrays, C

```
RED
GREEN
BLUE
BLACK
```

and D,

```
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
```

2 5 ↑ C implies the first two rows and five columns will be taken to form

```
RED
GREEN
```

3 ¯2 ↑ D takes the first three rows and the last two columns:

```
 4  5
10 11
16 17
```

0 4 ↑ A gives an empty array of shape 0 4! In any case, the shape of the result for a *take* operation A ↑ B is |A.

The *drop* operation produces a complementary result:

```
  3 ↓ A => 'DEFG'
 ¯3 ↓ A => 'ABCD'
  2↓ B => 3 4 5 6 7
 ¯9 ↓ B =>    (null vector)
 0 ¯3 ↓ C => RE
             GR
             BL
             BL
```

Can you see that the shape of the result of X ↓ Y will be 0 ⌈ (ϵY) − |X?

GRADE

A pair of monadic operators "grades" or sorts numeric vectors in ascending or descending order. A new development in APL extends these same operators to work dyadically on literal arrays, effectively giving the programmer control over alphabetic ordering as well as numeric. We will only take up the older monadic form here.

Suppose we have a vector:

A ← 10 ¯2 7 21 3 7 14.

Then the *grade up of A* (⍋A) will be 1 4 2 5 0 6 3, showing that the lowest value appears in A[1], the next lowest in A[4], and so on. Notice that where a value is repeated in the original vector, the result contains their indices in sequence (A[2] and A[5] both equal 7 here). Note also that this operation reflects the current index origin.

The *grade down* of this same vector [⍒ A) would be 3 6 0 2 5 4 1, ordering the indices of the values from largest to smallest.

RANDOM NUMBER GENERATION

Let me state immediately: The secret to writing sensible music lies in order, *not* randomness. The operators about to be described can indeed be useful to us, but not in a *direct* way such as choosing the notes of a melody!

Fittingly, the random number generator is designated by a question mark. If A is a scalar, the *roll function* (?A) will produce an integer chosen "randomly" from the set of integers in ⍳A. Here, A must itself be a positive integer. If A is any array of positive integers, the result will have the same shape as A and each entry will be selected independently as if ⍳A applied to each entry in A. (But remember: in reality, the index generator, takes only a scalar argument!)

The term *random* always requires explanation. APL incorporates an algorithm for picking random values, and that raises the obvious question as to whether numbers can be considered "random" if they are predictable. If you had an exact knowledge of the algorithm used here, you could write it in APL and execute it to see what the results of the ? operation would be. As long as you don't have such knowledge, the results will appear random to you.

But suppose you turn on your computer, perform certain operations, and then turn the computer off. Now you start up again and repeat the same sequence of operations. You will get exactly the same results if and only if the state of your computer is exactly the same at the beginning of each session. There is a system variable called the *random link*, ⎕RL, which is used by the algorithm in choosing random numbers; this variable is reset to a new value after each choice. You may save the current state of your APL "workspace" at any time so that, when you reload the workspace, the last value of the random link will be in effect. This means that the reloaded workspace will generate "new" random numbers even if the last session's input is repeated. Also, you can reset ⎕RL manually, so that you may control this aspect of randomness at will.

The dyadic random operation is called the *deal function*. In the expression A?B, both arguments must be positive scalar integers (or one-element vectors) and A must not be greater than B. The result will contain A different integers chosen from the set ⍳B. Note the difference between

? 5 ⍴ 7

and

5 ? 7.

The first case emulates five successive rolls of a seven-sided die, while the second deals five cards from a seven-card deck.

REVERSAL AND ROTATION

The symbol for the operations we're about to examine consists of the circle overstruck with the vertical bar (⌽. Consider the expression:

A ← ⌽ B

If B is a vector, Then A will contain the elements of B in reverse order. Thus ⌽⍳7 will equal 6 5 4 3 2 1 0. If B is a higher-order array, the axis operator is applicable. That is, we can write:

A← ⌽[N]A

where N is an integer denoting the axis along which reversal is to be performed.

And, as was the case for catenation, when N is the column (or last) axis, it may be omitted from the expression. Thus, if b is the two-dimensional array:

```
0 1 2
3 4 5
6 7 8
```

then **⌽[1]B** is the same as **⌽B** and equals:

```
2 1 0
5 4 3
8 7 6
```

while **⌽[0]B** will produce:

```
6 7 8
3 4 5
0 1 2.
```

In dyadic expressions, this operator produces cyclic rotations directed according to the axis operator and by an amount determined by the left argument.

2 ⌽ 'ABCDEFG' will equal **CDEFGAB**
¯2 ⌽ 'ABCDEFG' will equal **FGABCDE**

In general, **K ⌽ V** will cause a "rotation" K places to the left if K is positive and the same number of places to the right if K is negative.

Using the above matrix for B, **2⌽B** will equal:

```
2 0 1
5 3 4
8 6 7
```

¯2 ⌽B equals:

```
1 2 0
4 5 3
7 8 6
```

and **2⌽[0]B** would be:

```
6 7 8
0 1 2
3 4 5
```

In fact, the left argument may be an array whose rank is one less than the rank of the right argument, so that the columns, rows, planes, etc., can be rotated independently "around" the axis operator. Thus, **0 1 2 ⌽ B** gives

```
0 1 2
4 5 3
8 6 7
```

and **0 1 2 ⌽ [0]B** is

```
0 4 8
3 7 2
6 1 5
```

If, in the operator symbol, the vertical bar is replaced by a horizontal bar (⊖), the effect is to reverse the default axis operator from last to first. That is to say, instead of writing **⌽[0]A**, we can write **⊖A**. When the axis operator is explicitly given, both operators, ⌽ and ⊖, function identically.

A ⌽ [N]B equals **A ⊖ [N]B.**

TRANSPOSITION

The circle symbol overstruck with a reverse virgule or backslash signifies mathematical—not musical—transposition. To transpose a matrix, you simply interchange rows and columns, so that if A is:

```
0 1 2  3
4 5 6  7
8 9 10 11
```

then ⍉A is:

0 4 8
1 5 9
2 6 10
3 7 11.

Notice that the shape of A is 3 4, while th shape of the *transpose of A* is 4 3. In generalizing this monadic transposition to arrays of higher rank, it is logical to use this effect on the shape as a guide. Earlier implementations of APL made the generalization that the last two shape entries must interchange, so that an array with shape equal to 2 3 4 will transpose to one with shape 2 4 3. In other words, the planes would contain the same entries as before the transpose, but rows and columns would still interchange. Newer versions of APL generalize the change in shape as a reversal of the entire shape vector (⍴⍉A equals ⌽⍴A) so that two planes of three rows and four columns transpose to four planes of three rows and two columns.

The remainder of this section gives a more complete introduction to dyadic transposition. It is not a subject with immediate practical value in composition, so you may prefer to browse through it now and return for a more concentrated reading when you see a use for it.

Dyadic transposition is defined so that the left argument gives an explicit description of the way the axes to be reordered. In the expression P⍉A, P must be a vector satisfying these three conditions:

1) (⍴P = ⍴⍴A
There must be one element in P for each axis of A.
2) (⌈/P) ∊ ⍳⍴P
The greatest entry in P must be one of the integers in ⍳⍴P.
3) P must contain all the integers from ⎕IO to its greatest entry.

Thus, if ⍴⍴A equals 4, P can be any of the permutations shown in Fig. 5-4.

Taking the last case first in the figure (no repeated entries in the left argument), suppose the shape of A is 3 2 5 12. If P is 2 0 3 1, the shape of the result will be 2 12 3 5. To see this, write (⍴A) directly above P:

```
⍴A = 3 2 5 12
 P = 2 0 3 1
```

Then reorder the columns so the bottom row is in numerical order:

```
2 12 3 5
0  1 2 3
```

The top row now gives the shape of the result. This also says that the element in A[I;J;K;L] becomes the [J;L;I;K] entry in the result, so we can see in Fig 5-5 exactly how each element is affected.

In cases where P contains repeated elements, the same sort of [I;J;K;L] mapping can be used with modifications as needed for the duplicates. Say A

Values of P for ⍴⍴A = 4

0000
0001 or any of the other 3 permutations of these 4 elements
0011 or any of the other 5 permutations of these 4 elements
0012 or any of the other 11 permutations of these 4 elements
0111 or any of the other 3 permutations of these 4 elements
0112 or any of the other 11 permutations of these 4 elements
0122 or any of the other 11 permutations of these 4 elements
0123 or any of the other 23 permutations of these 4 elements

Fig. 5-4. Possible values of P in dyadic transposition for the expression ⍴⍴A.

element in A	position in P
[0;0;0;0]	[0;0;0;0]
[0;0;0;1]	[0;1;0;0]
[0;0;0;2]	[0;2;0;0]
.	.
.	.
.	.
[0;0;1;0]	[0;0;0;1]
[0;0;1;1]	[0;1;0;1]
[0;0;1;2]	[0;2;0;1]
.	.
.	.
.	.
[2;1;4;10]	[1;10;2;4]
[2;1;4;11]	[1;11;2;4]

Fig. 5-5. Effect of dyadic transposition using a ρρA value of 4 and a value of P having no repeated elements.

is the same as above, but P is now 1 0 1 2. As before, we can write:

3 2 5 12
1 0 1 2

and reorder to get

2 3 5 12
0 1 1 2

But, of the columns with duplicated P entries, only the column containing the smallest values will be kept. This means the result will have shape 2 3 12. Its effect on the lettered indices can be seen as:

I J K L becomes J I=K L
1 0 1 2 becomes 0 1   2

So the result will contain only the A[I;J;I;L] entries; the I and K indices here must always be the same, effectively eliminating one axis from the result. In the most extreme case, P=0 0 0 0, only the A[I;I;I;I] elements are used, forcing the result to be a vector with shape equal to the minimum entry in (ρA) or 2 in the above example, so that

0 0 0 0 ⍉ A

would contain only A[0;0;0;0] and A[1;1;1;1].

COMPRESSION AND EXPANSION

The symbols / and ∖ are used in two sets of operations. The first set consists of compression and expansion. These are dyadic operations in which the left argument must be a logical vector. The expression

1 0 1 1 1 0 1/'ABCDEFG'

causes the result ACDEG. Similarly,

1 0 1 1 1 0 1/117 96 34 102 84 57 91

produces 117 34 102 84 91. Where the left argument contains zeroes, the corresponding elements in the right argument are eliminated or *compressed out*. If the right argument is a vector, this forces the left argument to have the same shape. But if the right argument is an array of higher rank, the axis operator may be used and the left argument must be a vector with shape conforming to that axis. Suppose A is the array 3 4 ρ ι 12. Then 1 0 1/[0]A (or 1 0 1 ⌿ A) is

0 1 2 3

8 9 10 11

and 0 1 0 1/[1]A (or 0 1 0 1/A) is

1 3
5 7
9 11

Note, in the absence of the axis operator, the horizontal bar implies compression over the first axis. With neither the bar nor the axis operator, the last axis is compressed.

If either argument is a scalar, it will be reshaped to match the shape of the other as required. If both arguments are scalars, the result is either the null vector or a one-element vector.

Expansion is the logical antithesis to compres-

sion. Again, the left argument is a logical vector, but now conformability requires the number of ones in this vector equal the number of elements to be "expanded." That is,

```
1 1 0\'ABC'
```

gives AB C, inserting a blank in the thrird position, and

```
1 1 0\24 25 26
```

is 24 25 0 26. The axis operator can be used just as in compression:

```
1 0 1 1 1\[0]3 4 ⍴⍳ 12
```

equals

```
0 1 2 3
0 0 0 0
4 5 6 7
8 9 10 11
```

It may help to note that, in compression, the shape of the left argument conforms to the shape of (the axis used in) the right argument and the number of ones in the left argument equals the shape of the result (for that axis). In expansion, these relations are reversed—the number of ones in the left argument conforms to the shape of the right argument and the shape of the left argument determines the shape of the result.

Don't be misled by the direct use of logical vectors in the above examples. Remember, a logical condition such as V < 12 is a logical vector with the same length as the vector V. The statement

```
M ← (A[;0] ∊ 'XYZ')/[0]A
```

will keep all rows in the matrix A that have either 'X,' 'Y,' or 'Z' in their first column.

REDUCTION

Composite operations called *reduction* and *scan* also use the symbols / and \. To their right goes the usual numeric or literal argument. But, on their left can be one of the mathematical or logical operators! In practice, you'll find the most useful operators here are +, x, ⌊, ⌈, ∧, and ∨. For the moment, we'll restrict the discussion to the case where the argument of the operation is a vector.

The expression +/A represents the *plus reduction of A* or, dropping the APL jargon, the "sum over (the elements of) A." As instinctively clear as that may be, in order to understand how other operators can be used, we must analyze the way this expression is actually executed. Let A be the vector ⍳6. Then +/A represents

$$0+1+2+3+4+5.$$

APL must evaluate this from right to left as usual, though in this case it really doesn't matter. But now let's look at −/A! Here we have

$$0-1-2-3-4-5.$$

First, 4 − 5 equals ¯1. Then, 3 − (4 − 5) or 3 − ¯1 is 4. Following that with 2 − (3 − (4 − 5)) or 2 − 4, we find ¯2. And continuing in this way, we finally arrive at the result, ¯.

The logical operations AND and OR are particularly useful for determining the truth of a compound logical condition. The expression

```
∧/(A<B),(C=D),E∨F
```

will equal 1 only if all the conditions are true—A is less than B, AND C equals D, AND either of two logical variables, E or F, is true. In a similar way, an OR reduction will be true if *any* of the conditions in the argument are true.

The maximum and minimum operators will select the extreme values in the argument. The expression

```
⌊/31 47 26 70
```

will single out the value 26 as the minimum.

Be forewarned that the relational operators will not do what you might expect. For example:

=/A

will equal zero unless A has the form

1 1 1 . . . 1,a,a

Similarly,

</A

will equal one only if A has the form

0 0 0 . . . 0,a,b

where a is any number less than b!

Even more unexpected are the details of what happens when the right argument has only one entry or no entries. If the argument is a scalar or a one-element array, the result will equal that one element no matter what the operator. If the argument is the empty vector, the result will be what is called the *identity element* for that operator. This is inconsistent with the way APL treats other mathematical expressions in which the argument happens to be null. Both of these expressions:

5+⍳0
5*⍳0

give an empty result, but

5+ +/⍳0 and
5**/⍳0

equal 5! The identity elements can be thought of as the most "harmless" values for the operators. Can you see why the identity element for the maximum operator must be the smallest number that can be represented on your computer?

The reduction of a vector produces a scalar. In general, +/[N]A gives a result whose shape is the same as that of A with the [N]-axis removed. For example, with

A←3 4 ⍴⍳ 12

or

0 1 2 3
4 5 6 7
8 9 10 11

the expression

+⌿A or +/[0]A

will be the vector 12 15 18 21, while

+/[1]A or +/A

will be 6 22 38. Similarly,

⌈/A

equals a vector containing the largest entry in each row, while

⌊.0 /[0]A

picks out the smallest entry in each column. Notice the difference between

⌈/+/M and +/⌈/M.

The first of these totals the elements in each row of M and then picks the largest total. The second finds the largest element in each row and then sums over these elements.

Until these operations become second nature to you, there will be confusion about rows and columns. You may wonder if I referred to +/M as a sum over the elements in the *rows* by mistake. In the absence of the axis operator, operations are carried out *across* columns, i.e., across the last axis. Even the shape of an array is measured in this "orthogonal" way. That is, to count the number of rows, we actually count the number of elements in one column, and to find the number of columns we usually count across a row. The operator ⌽ rotates rows about the column axis while ⊖ rotates columns about the row axis. Once you have thought this through carefully, you will feel much more secure in reading APL expressions.

SCAN

The *plus scan of A* (+ \A) consists of a sequence of *plus reductions* producing a result with the same shape as A. If A is the vector 0 1 2 3 4 5, then + \A equals 0 1 3 5 10 15. The Nth term in the result equals the redution carried out over the first N elements. In this case,

+/A is equal to (+/0), (+/01), (+/01 2), (+/0 1 2 3), . . .

The extension to higher order arrays and to other operators is straightforward, but confusion will result if one forgets that *each of the successive reductions must be carried out from right to left*! To be sure you understand this, you should verify the following:

</ι6 equals 0 1 1 0 0 0
_/ι6 equals 0 0 1 0 0 0
≤/ι6 equals 0 1 1 1 1 1

INNER AND OUTER PRODUCT

The same mathematical and logical operators permitted in the reduction and scan operations are allowed in the two remaining composite functions—the inner and outer products. The *inner product* generalizes the mathematical operation commonly called *matrix multiplication.* It has the form:

A*a*.*b*B

where *a* and *b* are any of the allowed operators. For vector arguments, the operation is carried out as if it were written:

a/A*b*B

With the following conditions

.A←3 1 2 2 4
B←0 1 1 0

then A+,×B equals 7 (A×B is 0 1 2 0 4 and +/A×B equals 7), and A×.*B equals 8 (A*B is 1 1 2 1 4 and ×/A*B equals 8).

Conformability requires ¯1 ↑ ρ A (the last element of the shape of the left argument) be equals to 1 ↑ ρ B (the first shape element of the right argument). Scalars or one element arrays will be expanded to match as usual. The result will have the shape of the left argument with its last argument with its last element dropped, catenated to the shape of the right argument without its first element. If A and B had shapes 5 2 2 3 and 3 6 4 respectively, the result would have the shape 5 2 2 6 4. Note the "conforming" axes in the shapes are eliminated in the result. (This is why, when both arguments are vectors, the result is a scalar.) Any particular element in the result can be found by the expression

R{I;J;K;L;M] equals *a*/A[I;J;K;] *b* B[;L;M]

The outer product operation causes all the members on two variables to interact through a single operator in a specific order. In the expression

R ← A∘.+B

(read: "R is set to A jot dot plus B"), the result contains every element of A added to every element of B. The shape of the result equals the catenation of the *shapes* of A and B. Each element of the result can be determined directly from its indices, e.g., if A has rank 2 and B has rank 3, then

R[I;J;K;L;M]

equals

A[I;J] + B[K;L;M]

probably the most useful situations occur with two vectors, Say

A←5 4 3 2

and

B ← 1 2 3 4 5 6.

Now, A°. +B is the array:

```
5 6 7 8 9 10 11
4 5 6 7 8  9 10
3 4 5 6 7  8 9
2 3 4 5 6  7 8
```

Here, each row index corresponds to the entry in A with the same index and each column index has the same relationship to B. Then A°. >B equals

```
1 1 1 1 1 0 0
1 1 1 1 0 0 0
1 1 1 0 0 0 0
1 1 0 0 0 0 0
```

A°. =B produces

```
0 0 0 0 0 1 0
0 0 0 0 1 0 0
0 0 0 1 0 0 0
0 0 1 0 0 0 0
```

and B°. ⌊A is

```
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 2
4 4 3 2
5 4 3 2
5 4 3 2
```

BACK TO NUMBERS

Different computers have different "architecture" and therefore different limits on the size of the numbers they can handle. The system programs that allows different computers to execute APL programs are also different and may impose additional restrictions on numeric values. If you were to enter a number such as:

1.23456789012345678

the computer might "answer"

1.23456789

The number of print positions used for the displayed value depends on the setting of a system variable called *printing precision* (□ PP). This can be changed from its default value in some systems to any integer up to about 15. A slightly larger number of digits than the number displayed is maintained for calculations. But if this limit is violated in your program, your results will be rather upsetting. For example:

123456789012345678 − 123456789012345667

clearly has a value of 11. Yet a computer will probably tell you it equals zero!

On entering a number such as 123456789012345678, the response will take this form: **1.23456789E17**. This is a common form of exponential notation used in higher-level computer languages to keep track of the position of the decimal point. A number followed by E17, means that the number is to be multiplied by 10 raised to the 17th power. That is, the decimal point is to be moved 17 placed to the right. Similarly, **1.2345E¯9** would represent the value 0.0000000012345 (the decimal point has been moved nine places to the left).

In most applications, and certainly in composing, we never deal directly with such extremes in value. However, there may be times when numerical artifacts introduced by a calculation will cause real problems. This usually happens when you test for a particular relation between two values that are not necessarily integers. You may be waiting, typically, for the variables A and B to become equal, and they never do. After "tracing" the execution of your program very carefully, you find that they approached each other to within some number containing the exponent "E¯13"! Another system variable, *comparison tolerance* (□ CT) can be set to get around this. Instead of using the relation A = B—where you expect a zero or one and the "true" condition never appears—you can write the relation as

□ CT ≥ | A − B.

If the default value of "quad **CT** (usually around 1E¯13) is unnecessarily small, you might reset it to a more reasonable value for your purposes, perhaps 1E¯6 or even 1E¯13 3

6 Scoring in APL

You have seen how most of the pieces move in this game. In this chapter we'll look at some additional features that serve as strategic tools. The most important of these give you the ability to define your own "pieces" and their "moves." And once more, "APL-speak" tends to fuse the ideas of pieces and moves into one concept: user-defined functions. But before going into this, let's look more closely at the "board" on which this game is played. Much of the material we will discuss here applies to the direct use of APL on a computer, and so may not appear immediately useful if you plan to code in some other language.

Unlike computer languages that only allow you to create and execute programs, APL lets you establish environments called *workspaces*. When you tell your computer that you wish to use APL, it places you in the default environment, called a *clear workspace*. Also, it places you in execution mode, meaning that any expressions you enter will be immediately executed. Of course, you must switch to definition mode to create functions of your own, and we'll discuss this below. But anything you do in the workspace will be lost if it is not saved. And before a workspace can be saved, it must have a name other than CLEAR. So, in the name of clarity, we must examine some of the system commands before going any further.

SYSTEM COMMANDS

Statements that begin with a right parenthesis, relate directly to the APL environment or workspace. These are called *system commands*, and here we'll describe some of the most common forms.

)WSID. In response to this command, the computer will display the name of the currently active workspace.

)WSID MUSIC. This command changes the name of the current workspace to MUSIC. Of course, any name valid in your system may be used. Most systems respond by showing a message such as "was clear" or "was harmony" as an aid or warning.

)SAVE. The active workspace will be saved on peripheral storage (such as tape or disk) when this command is given. If a workspace with the same name already exists in the user's library, it will be replaced. Typically, a message will signal a successful operation by telling you the name of the current workspace and the time and date. Less favorable messages such as "**not saved—disk full**" or "**not saved—clear workspace**" may appear. In some systems, the command can contain a (possibly different) name under which the workspace is to be saved, such as)SAVE OPUS6.

)LIB. Here the user will be shown the names of his saved workspaces. There are variations of this command, such as)LIB 32 or)LIB M, which give you access to particular libraries or workspace names that begin with certain letters, but these depend on the way your system handles libraries.

)LOAD ORKSTR8. A copy of the workspace ORKSTR8 becomes the active workspace with this command. On completion of the loading process, a typical message here would be

SAVED 2/30/85 02:14

meaning that this workspace was last saved at a most unlikely date and time. On the other hand, a message such as

WORKSPACE ORKSTR8 NOT FOUND

might appear if our spelling or memory for names is faulty.

)COPY ORKSTR8. The workspace ORKSTR8 is added to the active workspace when this command is given. This makes all user-defined variables and functions in that workspace available in addition to whatever is already active. However, this statement will cause entities in the active workspace to be replaced if their names are the same as those in the copied workspace. To prevent this, the protective copy command,)PCOPY ORKSTR8, can be used to bring in only those objects from ORKSTR8 whose names do not already exist in the active workspace. It is important to remember that system variables (such as ⎕ IO) are not brought into the active workspace with the)COPY command, but are with)LOAD.

)COPY ORKSTR8 BRASS REEDS. This more restrictive form will copy only the objects name BRASS and REEDS from the workspace ORKSTR8. The PCOPY form is also available in some systems.

)DROP BEATS. The workspace named BEATS will be erased from the library with this command.

)CLEAR. The active workspace can be replaced by the default environment at any time. All user-defined functions and variables will be lost and all system variables will be reset to their original values when this command is given.

THE FUNCTION HEADER

To enter or leave definition mode, the *del*, ∇ , (not to be confused with delta, ∇) is used. If you are entering this mode to define a new function, the del is the first character in a *header statement*. The most general form for a header would be

∇A ← B BLEEP C;D;E;F . . .

Besides switching to definition mode, this particular statement says a dyadic function named BLEEP (with left and right arguments B and C) is about to be defined. This function will also return an explicit result, specified here by A. Further, certain *local variables* (D, E, F, etc.) will be used within this function.

When this function has been completely defined and we are back in execution mode, the name BLEEP will be available for use as if it were another APL operator. The statement

X BLEEP Y

will cause the explicit result to be displayed because no assignment (with a left arrow has been made. In a more involved setting,

B ← N × (R BLEEP J*2) * .5

will execute the function using R as its left argument and the square of J as its right argument. The square root of the result will be taken (result raised to the 0.5 power), multiplied by N, and the final

value will be stored in the variable B.

No doubt you have noticed that the left and right arguments used in the header, as well as the name of the returned result, are all "dummies." Within the body of the function, these names must be used consistently, but when the function is called, any valid expressions will serve as arguments. Validity here refers not only to APL syntax, but also to the requirements set by the definition of the function. That is, if the function expects the left argument to be literal and the right argument to be an array of integers of rank three, any deviation from these requirements will cause execution to stop as soon as an illegal operation is attempted.

Note also that no reference is made to the local variables when *calling* the function for execution. Local variables have no effect upon like-named variables in existence when the function is entered or after execution of the function. In fact, when one function calls another function, local variables in the calling function will appear as *globals* to the called function unless its header declares the same names are to be used for its own locals. Such names serve more for efficiency than utility. If you don't use local variables in the header all variable names that appear in the function, except the arguments and assigned result, will be treated as global variables. There is nothing wrong in this, but global variables add to the permanent burden in your workspace. There is also something called a *symbol table* that contains all the names you use, local as well as global. By using locals with the same names in different functions wherever possible, you reduce both of these loads and help avoid two rather unpleasant messages: "workspace full" and "symbol table full."

All names used in a header statement must be different, but this does not apply to the expressions that call a function. For example,

```
E ← F BLEEP E
```

will use the current values of the variables E and F on entering the function and, on leaving, store the final result in E, replacing its original contents.

The simplest possible form for a header statement is

```
∇ GT2XV
```

where there are no arguments, no explicit result, and no local variables. This function will do its thing whenever the user enters GT2XV. Between these two extreme forms for a header are all the various possibilities that can be constructed from the three considerations.

1. How many arguments (0, 1, or 2)?
2. Is there an explicit result?
3. Are there any local variables?

The answer to the first question determines how the name of the function will appear in the header as well in any expression to execute the function:

```
NAME
NAME ARG1
```

or

```
ARG2 NAME ARG1
```

The second question determines whether an assignment is to appear in the header (∇ RESULT ← . . .) and whether the call statement can use the name of the function as if it represented a (numeric, literal, or logical) value. As we have seen, the third question has no effect on the statement that calls the function for execution.

WORKING IN DEFINITION MODE

After you have entered the header statement, the computer will say it is in definition mode by displaying [1] at the beginning of the next line. The next move is yours, and the signal telling you this will probably be a blinking cursor on your display screen immediately following the right bracket. On the same line, you must now enter the first statement of your function. When that is done the computer will display [2] and this process will continue until you enter a del character, either by itself or as the last character of a statement. Definition mode ends, at that point you are again in execution mode, and the function just defined is available for execution.

While in definition mode, it is easy to make typing mistakes and equally easy (or nearly so) to correct them. But the ways this might be done vary with the capabilities of different APL implementations. Most systems have the old "del-editor" as well as newer, display-oriented, full-screen editors. In any case, you'll find there are ways to insert and delete as well as change statements in a function. And, as the term *editor* suggests, after defining a function, you can get back into definition mode to make changes.

This is done by typing the del character followed by the name of the function without any other header information. In entering the del-editor this way, the computer will assume you want to add new information to the "bottom" of your function, so the next available line number will appear in brackets and you'll find yourself back in definition mode. If you reenter definition mode by typing the del character, the function name, and then a bracketed quad symbol, so:

Δ *function name* [□]

the entire function will be displayed, followed by the next line number in brackets, and a blinking cursor waiting for you to continue the definition. By including a line number inside the brackets before the del, you tell the computer to display that statement alone and then go into definition mode at that same line number, allowing you to alter or reenter the line as described. Of course, you need not do so; you might type some other bracketed information to display or move to another part of the function, or simply enter another del to return to execution mode without changing anything at all.

Typing a del before and after this kind of statement:

Δ *function name* [6 □] ∇

causes the indicated line to be displayed, while

Δ *function name* [□] ∇

displays the entire function. In either case, after the display is complete, *execution* mode is in effect! It works as though the first del takes you into definition mode and the closing one takes you out.

Clearly, we can't go into all the mechanical aspects of defining and editing functions here without choosing a particular system for demonstration. But you will find such information readily available when you have access to a system, and it is easily and quickly learned. Of far greater importance for what follows are logical considerations that arise when you define functions, and to these we must now turn.

BRANCHING AND LABELS

In execution mode, we can only tell a computer to do things in sequence: do this, now do this, etc. The main reason for the existence of functions is to permit a collection of previously entered statements to be executed with a single command. But within a user-defined function, we also need the ability to break the ordinary step-by-step sequence of operation. This requires that we be able to direct the computer to execute a line of code other than the very next one in the function; we introduce the right arrow (→) branch symbol.

The most direct way to force execution to deviate from the sequential path is with an expression of the form

→ *n*

where *n* is the (unbracketed) line number of the statement to be executed next. But forget it! It is too easy to overlook such a statement when you edit a function, inserting or deleting lines so as to cause the remaining lines, including the one to which you want to branch, to be renumbered. You might even trap yourself in an infinite loop:

[6] → 6

Statements can be labeled in order to avoid this problem. *Labels* have characteristics much like those of local variables (actually, local constants), so you must choose names for labels that don't conflict with any existing variable names. They will appear in your programs in just two ways:

1) To label statements

2) As the destination fo branching instructions

A label is set off from its statement by a colon:

```
LABEL1: X ← X - 2
```

Clearly, renumbering the lines after editing the function will not effect the label or the statement that branches to it:

```
→ LABEL1
```

but branching statements are not restricted to such a simple form.

Any expression can be the argument of a branch symbol as long as it yields a label within the function or a numeric vector which is either null or has an integer as the first entry. Labels might appear as elements of a vector in an expression such as

```
→ (CALC, SETUP, 0, ERROR) [J]
```

or

```
→ LOGVEC/CALC, SETUP, ERROR.
```

In the first example, if J is zero execution will take us to the statement that begins with

```
CALC: • • • .
```

If J were equal to two, the statement would effectively be

```
→ 0
```

This is taken to mean that execution of the function is complete and the computer is to procede from the point where this function was called. A branch to any integer outside the range of the line numbers in the function will have the same effect.

In the second example, **LOGVEC** is a logical vector (all ones and/or zeroes) and the first label appearing in the compressed result is the cone used. If **LOGVEC** contains only zeroes, the compression yields the empty vector. This is the same as →⍳0 and means that the very next statement should be executed—no branch at all! Contrary to what you would expect, this is not the same as a right arrow with no argument following it. The statement containing the single symbol, → , commands an immediate halt to execution, as though the conductor put down his baton and walked away!

Another kind of branch—the Hamletic (to branch or not to branch)—frequently will appear in the demonstration programs of later chapters:

→*(condition)* ↑ LABEL.

If the condition is true, its value equals one, and the branch occurs. If the condition is false, the *zero take* causes a branch to the empty vector which we saw above is no branch at all. There will be occasions when the branch should be taken only if the condition is false. For that situation we can use either

→ (~*condition*) ↑ LABEL

or

→ (*condition*) ↓ LABEL.

EXECUTE

You are now aware that APL statements produce results that are either numeric or literal. Even the empty vector is either numeric (⍳0) or literal (''). Yet, in user-defined functions, we can make good use of two powerful operators to convert from one form to the other in surprising ways. The "⍎" is used monadically to execute a literal scalar or vector argument, just as if that argument were an APL expression. This requires that the argument can indeed be evaluated to form a legitimate expression. For example,

```
⍎ 'A ← B'
```

is totally equivalent to

```
A ← B.
```

But suppose A is the literal vector **'SCALE'** and B is the numeric vector **0 2 4 5 7 9 11**. The expression:

⍎ A, '← B'

catenates the literal string 'SCALE' to the string '← B' and then executes the complete expression

SCALE ← B

effectively creating the variable named SCALE containing the same numeric values as the vector B.

More complex expressions can decide just what is to be executed based on some conditional relation between variables.

R ← ⍎ 3 ¯5 [I = J] ↑ 'A + A +\A,B'

will calculate the sum of A and B if I and J are not equal. This is true because the bracketed quantity I = J evaluates to zero, selecting the 3 from the two-element numeric vector, and then "taking" the first three elements from the literal vector for execution. This results in the expression R ← A + B. If I does equal J, the last five entries in the literal string are executed, so that R is set to the *sum scan* of A catenated to B.

In our programs we will make use of this extremely powerful (and equally dangerous) expression:



This says, "Execute whatever the user enters." The obvious danger lies in the ease with which one can type an APL solecism, causing the program to stop and display an error message which the user may not understand if he is not the programmer. Actually, we will set a variable equal to quad-quote (X← ⍞) and, after examining X for certain special characters, decide whethr or not to execute X (⍎X).

Newer versions of APL have a system function (⎕ EA) which acts as a dyadic execute function. The right argument is used exactly as with the ⍎ operator. But if an error occurs, instead of stopping with an error message, the program executes the alternate left argument! (The "EA" stands for execute alternate.) Only if the left argument is also in error will the program stop.

FORMAT

When used monadically, the ⍕ symbol transforms a numeric argument to a literal one with exactly the same appearance. It leaves literal arguments unchanged. The statement

'3 times 4 equals ', (⍕3 × 4), '.'

will cause the following line to be displayed:

3 times 4 equals 12.

If A were the array 3 4 ⍴⍳ 12 and B was set to ⍕ A, both arrays would print as:

```
0   1   2   3
4   5   6   7
8   9  10  11
```

But, while the shape of A is 3 4, the shape of B would be 3 12 because each number is "formatted" to match the spacing in the original (numeric) array.

When used dyadically, the right argument of this function must be a numeric array. The left argument may take on various forms, depending on the system being used, in order to give you precise control over the way the right argument is to be displayed. In newer systems, the left argument can even be literal so as to describe graphically how the numbers in the right argument are to be formatted. As such, this is a far more useful tool for an accountant than a composer, so further elaboration will be given only if needed in any of the musical examples.

TRACE AND STOP CONTROL

Too often our most carefully planned programs don't behave exactly as we expect, and we must face up to the task of debugging them. Suppose that in such a misbehaving program a function named GENER8 is suspect. An expression such as

T ∆GENER8 ← 3 4 12 17

establishes a special *trace-vector* containing the integers resulting from the expression following the assignment arrow (here, 3 4 12 17). These should correspond to line numbers in the function. After this vector is defined, the function can be executed

in a normal way. But as each of those particular lines in the function is evaluated, the line number and the value(s) resulting from the corresponding expression will be displayed:

```
GENER8[3]7
GENER8[4] 0 0 1 0
GENER8[12] G B C# D etc.
```

By resetting the trace-vector to zero or iota zero, tracing can be removed.

Although the trace vector appears to be an ordinary variable, it is not. The form T Δ **NAME** can only appear to the left of an assignment arrow, but can be used within functions as well as directly in execution mode. It cannot undergo any sort of test that could be applied to regular variables.

The *stop-control* has identical attributes. Its form differs only in the leading character being S instead of T:

SΔGENER8 ← *expression*.

During execution, when a line number is reached that exists in the stop vector, that line number is displayed and execution stops *before* that line is executed. For example, if the expression contained a 6, when that line was reached, we would see

```
GENER8[6]
```

and the computer would wait to execute any commands we enter. At this point, we could examine the values of any variables—even local variables, because we are in execution mode *within* that function. We might ask to see the expression constituting that line of the function:

```
∆GENER8[6 ⎕ ]∇
```

and then execute parts of that expression "manually" to see what's causing our problems.

To return to execution from the stopped state, we enter a branch statement to take us right back into the code. In this case, →**6** or →⎕ **LC** would resume execution at line six. (Quad LC is a system variable—the line counter—pointing to the number of the line most recently set for execution.) Of course, we could branch to some other line or even branch out of this function. And, as with the trace vector, stop control can be removed by setting the vector to zero or empty.

STATUS INDICATOR

When execution is halted, either because of stop control or an error in a statement, you can display the "status indicator" to see where the program is stopped and how it got there. That is, suppose an error occurs and execution stops with the message

```
VALUE ERROR
ZZ[3]     TM ← B[1;]/LCO[;K]
                    ^
```

The message and caret indicate that values have not been assigned to the object named **LCO**, so the program cannot execute the third line of the function, **ZZ**. Now, if you enter

```
)SI
```

a display of the following sort will appear:

```
ZZ[3]
GETL[7]
CUTOFF[11]
START[6]
```

This shows that statement number seven in the function named **GETL** called the function **ZZ** and is waiting for that function to complete its task. Similarly, execution of the function **CUTOFF** is pending at its eleventh line for completion of **GETL**. And **CUTOFF** was called by the sixth statement in the very first function of the program, **START**. Such information should refresh the programmer's memory concerning where in the sequence of events the variable **LCO** should have been assigned values. While the halt is in effect, you can examine the various functions,

```
ΔGETL[ □ ]∇
```

and even execute these or other functions. With care, you can also make changes in various functions or variables. For example, you may see that you simply forgot to initialize the variable LCO. Then you could enter the statements:

```
LCO ← 3 4 ρ 0
→3
```

Execution will then resume until another stop occurs or until the program runs to completion. Suppose however, you changed something in the program while it was halted in such a way that the computer's internal bookkeeping on the sequence of events was no longer valid. That is, if instead of the direct initialization of LCO you had entered:

```
∇ZZ[2.5]     LCO ← 3 4 ρ 0 ∇
```

thereby inserting this statement just before the one that caused the error report, another message would appear:

```
SI DAMAGE
```

This says that the existing pointers to lines in the functions are no longer valid so that execution cannot resume. The status indicator must now be cleared either by entering a solitary right arrow or by reloading the workspace.

By executing other functions while one is halted, you could cause other stops to be encountered. If this happens, the)SI list will be lengthened. The most recently halted function will appear first, marked with an asterisk, and followed by the functions pending its completion. Immediately following these will be the next halted function (with an asterisk) followed by its pending functions, and so on. Before any lower-level execution may be resumed, the ones above it in this list must be completed or cleared with the right arrow.

A workspace can be saved with halted functions so that, on reloading, you are back at the point where the stop occurred. This is a convenient way to carry out a long session. It is also an easy way to cause perplexing problems if you forget that there are halted functions; local variables that are in effect will "shadow" globals with the same names, so that variables whose values you wouldn't think to question can make your computer appear to have gone crazy!

SYSTEM FUNCTIONS

There are a number of system functions which, like system variables, begin with the quad symbol. While most of them are quite useful, only the three that can be used directly to create or annihilate functions will be mentioned here.

The *canonical representation* function (□ CR) can duplicate a function in the form of a literal matrix. If we set:

```
M ← □ CR 'FNM'
```

then M will be a character array that looks just like a listing of the function named FNM, except that the leading and trailing del symbols and the line numbers will be missing.

An inverse function to quad CR is the *fix* function (□ FX). Suppose, after executing the above expression creating M as the canonical representation of the function FNM, we altered M in some way. If we then enter

```
K ← □ FX M
```

the contents of M will be copied back into the form of a function. Assuming we did not alter the name FNM in the first row of M, the function FNM will be replaced by the form it now has in M, and K will be set to the literal vector 'FNM'. Had we changed the name in M[0;], a new function would be created with the new name, and K would be set to contain that name as a literal vector. If for any reason the matrix cannot be converted to a legitimate function, K will be set to the index of the row in M that contains the problem.

The last system function to be mentioned here is *expunge* (□ EX). This function will remove

variables or functions (that are not halted functions) from the workspace. The expression:

K ← ⎕EX 'FNM'

will try to erase the object named **FNM**. If successful, the value one will be returned in K, otherwise K will equal zero. If **NAMES** is a literal matrix containing one variable or function name in each row, the statement:

⎕EX NAMES

will return a logical vector in which the ones and zeros apply to the corresponding rows of the matrix, signalling whether or not the named item in each row was successfully purged from the workspace.

There is no reason why a matrix can't be created by a function so as to serve as a canonical representation of a new function. The function that creates the matrix can fix it (⎕FX), execute it, and then expunge it! This is a rather sophisticated technique, and I close this discussion on such a note just to point out that, as in any natural language, although the vocabulary is finite, there are no limits to the number of things that can be said or to the number of ways any one thing can be said.

7 Rhythm

Of the three primary components of Western music—rhythm, melody, and harmony—rhythm is the one most readily represented in a formal framework of numbers. Standard musical notation attempts to provide a representation through symbols, but it has unavoidable deficiencies because ordinary symbols don't carry the well-defined logic of the symbols we call numbers.

MUSICAL NOTATION

As a quick refresher for those who haven't dealt with this subject since grade school, we can describe the system in use in terms of circles and lines. To begin, we define an open circle (𝅝) to have the value one, and call it a *whole note.* By attaching a vertical line

𝅗𝅥 or 𝅗𝅥

we reduce its value by half (to one-half) so that a *half note* fills a duration half as long as a whole note. Some temporal unit is arbitrarily chosen for reference so that the values one and one-half are in some way made relative to that selected unit of time.

Now, by filling in the circle, we reduce the value by half again, so that

𝅘𝅥 or 𝅘𝅥

are called *quarter notes.* From this point on, the halving process continues by adding short lines to the vertical line:

𝅘𝅥𝅮 equals an *eighth note*

𝅘𝅥𝅯 equals a *sixteenth note*

and so on. A sequence of such notes is extremely difficult to read unless they are grouped together so that the individual beat durations stand out visually. This is the same situation pointed out in connection with binary numbers; four groups of four bits can be grasped mentally with ease, while

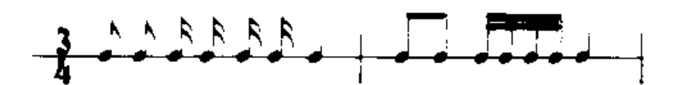


Fig. 7-1. Two measures containing identical musical information. The second bar is easier to read because the notes have been grouped to show the three beats.

an unbroken sequence of 16 binary digits is as legible as asentencewithoutspacesbetweenwords.

By way of demonstration, the two measures shown in Fig. 7-1 have exactly the same rhythm. Only in the second measure are the beats and their subdivisions obvious. In measures such as these, that are to have three beats, instead of letting a whole note occupy one bar and defining a symbol to represent a "third note" which would fill each beat duration, we declare the *meter* by writing a "3" over a "4" (3 × 1/4) and call this *three-four time,* meaning there are three beats in a measure and a quarter note fills one beat. Though correct, we would not say the duration of one measure is three-quarters of a whole note. Describing the meter in this way, and then applying some temporal reference unit to a whole note (e.g., 𝅝 = 1.74 seconds), simply because that note has the value one, may be logical, but it's most inconvenient. Instead, a tempo is assigned in terms of a beat frequency. The notation

♩ = 138

implies a tempo of 138 quarter notes per minute.

Ironically, the primary deficiency in this system relates not to the number one as the value of a whole note, but to the number two as the only implicit divisor relating one symbol to another. Durations of odd length must be expressed by "tying" notes together. For example,

𝅗𝅥‿♩

has the duration of three quarter notes, while

♩‿♪

is the equivalent of three eighth notes in duration. An abbreviated form using dots makes these particular values easier to write:

𝅗𝅥. and ♩.

are equivalent to the two cases above because a dot increases the value of a note by half its ordinary (undotted) value. But this is a half-hearted attempt to supply the missing values in such a system. There are no unique symbols for durations such as a third, a fifth, a sixth, a seventh, a ninth, etc.

A feeling of subdividing beats by three rather than two can be imparted through the use of meters with six, nine, twelve, etc. "beats" in a measure. Notice, a bar of three-four time and a bar of six-eight have the same number of eighth notes (Fig. 7-2) but in the first measure we feel three beats equally divided in two and in the second we experience two beats divided in three. When the upper number in the time signature is divisible by (but not equal to) three, we are dealing with a *compound* or *triplet* meter and usually we group the notes as shown in threes. The misleading number of "beats" announced by the time signature reflects the restricted use of numbers in this symbolism. When we try to say there are two beats in a bar and *three* eighth notes get one beat, we need to recognize this is 2 × 3/8, or 6/8.

A more insidious problem can be traced to the fact that aural taste is as susceptible to addiction as oral! When a composer grows accustomed to using triple meters, he begins to "need" groups of three subdivisions in a beat as much as he does two or four. And when composers first found they could freely mix double and triple meters mentally and instrumentally, a way had to be found to do so symbolically. The solution was to introduce so-called



Fig. 7-2. Two measures with different meters but with the same attacks and durations.

"artificial groups" to account for these quite "natural" rhythms. Thus Mozart was able to write rhythms such as the one shown in Fig. 7-3. Here, the digit 3 under a group of three notes indicates they are to occupy the same duration as an ordinary group of two such notes. Until recently, there had been so little experience with quintuple and septuple meters that practically no consideration has been given to the need for distinguishing between groups of five in the time of four and five in the time of three or between seven in the time of six, or in the time of five, or four, or three!

A notation system, like a written language, should let a composer express anything he wishes. But, what a composer wishes to say depends on his ability to think deeply in that "language," and to express his thoughts as simply as possible in "writing." Idiomatic phrases, common expressions, and clichés tend to form the major part of everyday "conversations"—musical as well as verbal. When something new is said, it often requires extended explanation.

A common musical idiom is jazz, and the conventional notation system is used "idiomatically" in jazz. That is to say, when a composer writes a rhythm such as the top line of Fig. 7-4 for a jazz musician, he expects (and usually gets) something closer to the one on the bottom line. If he were to write the first form for any of today's computers that "read" music, he would be most disappointed by the performance.

A common musical expression is shown in the first bar of Fig. 7-5 but not so common is the second measure. There is no way to explain such a rhythm other than by playing it repeatedly until it *makes sense* to the listener. All rhythms based on what are called "rational numbers" are inherently "sensible" but, until now, composers had to discover new rhythms the hard way—i.e., by learning to hear and play them without benefit of an instructor. Composers, musicians, and ordinary listeners are all strongly influenced by their musical experience, and musical experience is anchored to the notation system. So here we have a "negative feedback loop" that tends to discourage discovery and acceptance of anything that has not been written in this system or cannot be so written. Today, rhythms can be described in ways that will let a computer "explain" them! As proof, I will present a program that can be run on certain computers to play any rhythm you can conceive. But before doing so, I must introduce someone who laid the groundwork for all that follows.

THE WORK OF JOSEPH SCHILLINGER

Some two decades B.C. (before computers), Joseph Schillinger was teaching artists and musicians a methodology which is quite applicable to computers and synthesizers today. In fact, his soundness of mind was often in question because of his insistence that the day was not far off when machines would be able to compose and perform music and "plot" works of art. Of course, at that time his methods could not depend on such devices. His fundamental axiom stated the need for

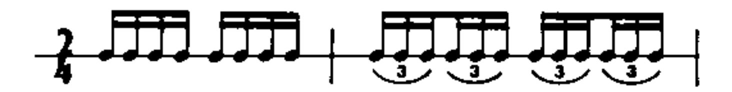


Fig. 7-3. Mixing double and triple meters through the use of artificial groups.
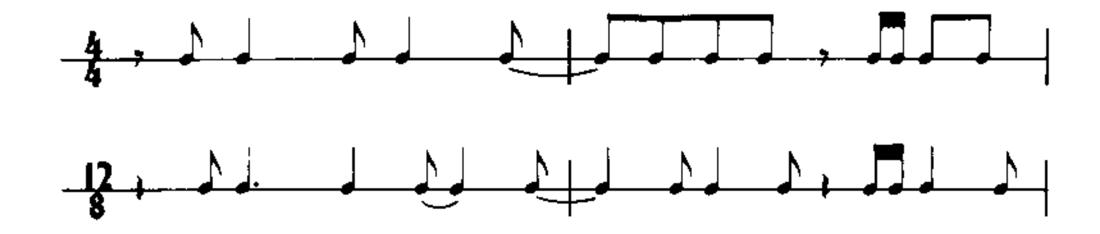


Fig. 7-4. The first line shows a rhythm written for jazz performance, while the second illustrates how it probably would be interpreted and played.

"regularity and coordination" in works of art, and his theories were based on the use of "natural numbers" (meaning integers) in order to achieve them. From this developed his "Theory of Rhythm."[1]

The rhythm shown in the first bar of Fig. 7-6 can indeed be expressed as the sequence of integers

4 2 2 1 1 1 1 4

where each number represents a duration in units of a sixteenth note. That is, 1 implies one sixteenth note, 2 implies an eighth note, 3 implied a dotted eighth, 4 implies a quarter note, 5 implies a quarter note tied to a sixteenth, etc. We can also express this sequence in a way that groups the subunits in terms of (quarter note) beats:

1(4) 2(2) 4(1) 1(4)

Here, we see two "rhythms," the original rhythm of durations, and a rhythm of attacks (1 2 4 1).

But, to increase our faith in Schillinger's assertion about the adequacy of integers to express "regularity," let's consider a truly pathological case—the second measure of Fig. 7-6. The first note here is clearly one eighth. A little thought will show the next three are "twelfth" notes (because 12 of them would equal a whole note). This same sort of reasoning shows the next artificial group has five "twentieth" notes, and the last has seven "twenty-eighth" notes. Had Schillinger settled for rational numbers rather than integers, we could be satisfied now by expressing this rhythm as:

1/8, 3/12, 1/8, 5/20, 7/28.

But for an additional step to prove that an integer representation exists, we need only put all these fractions in terms of a lowest common denominator. That denominator is then our *unit* and the numerators are the integers that validate Schillinger's claim. After a bit of multiplication, I find this rhythm to be

105 70 70 70 105 42 42 42 42 42 30 30 30 30 30 30 30

in units of one 840th of a whole note. Using coefficients that show an attack rhythm, we have

1(105) 3(70) 1(105) 5(42) 7(30)

where the groups themselves have durations 1 2 1 2 2 in eighth-note units!

The point to be made of all this is more subtle than the fact that the temporal forms of regularity

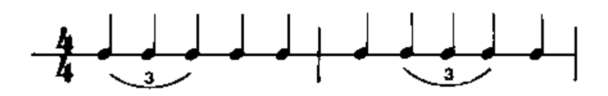


Fig. 7-5. The first measure is a common rhythm using quarter-note triplets; the second measure shows an unusual application.

[1]J. Schillinger, *The Schillinger System of Musical Composition*. New York: Carl Fischer Inc., 1946. *The Mathematical Basis of the Arts*. New York: Philosophical Library, 1948.



Fig. 7-6. The first measure shows the rhythm of durations 4, 2 2, 1 1 1 1, 4. The second represents the rhythm 105, 3(70), 105, 3(70), 105, 5(42), 7(30).

we call rhythms can be represented by integers. Of greater significance is the idea of *equating* a rhythm with a string of numbers. This extends the notion of rhythm in a remarkably firm way into dimensions other than time. That is, just as attacks can replace durations as the basis for a rhythm, so can any other countable or measurable quantities. Thus we can describe rhythms of length, angle, pitch, and so on.

An interval rhythm based on the original sequence of durations, 4 2 2 1 1 1 1 4, would be (C) E F♯ A♭ A B♭ B C E. Here, measurements are made in semitones beginning with the reference pitch in parentheses. We might superimpose a directional rhythm on this, requiring successive intervals to be measured two upward, one downward, two upward, one downward, etc., producing the result:

(C) E F♯ E F F♯ F F♯ B♭ F♯ A♭ B♭ A B♭ B B♭
D F♯ E F♯ G F♯ G . . .

In effect, we've superimposed a ryhthm of "+ + −" on "42211114" giving the resultant sequence:

4 2 −2 1 1 −1 1 4, −4 2 2 −1 . . .

The three-element rhythm of signs will need to be repeated eight times, and the eight-element interval sequence three times in order for the two to "come out even." But even then, the pitches produced may not have cycled back to the starting point, so the whole rhythmic pattern could repeat and still generate "new" material.

A further step in the direction of increasing abstraction would take us from measurements (intervals) to scale entries themselves. The entries in a scale can be markings on a ruler or other measuring device; a sequence of colors, shapes, or sizes; or in the case of music, the notes in any pitch scale or tuning system. Using the tuning scale

C C♯ D E♭ E
0 1 2 3 4

our "rhythm" (42211114) becomes

E D D C♯ C♯ C♯ C♯ E

Just as we can measure or count with a scale, we can enumerate and even order the kinds of scales we use. And once more, "scale" should be interpreted in its broadest sense. Frequently, the spatial directions *x, y,* and *z,* which normally measure from east to west, south to north, and vertically upward, are numbered 1, 2, and 3 by mathematicians. A scientist working in "thermodynamic space" might replace those "normal" measurements with pressure, volume, and temperature. In this sense, a composer works in musical space with scales of pitch, time, volume, timbre, and so on. A generalized abstract space may also include mathematical, physical, and logical *operations* as directions! Specifying a point in such a space by coordinates (2, 3, 4, 7) might mean: Move the given object two units to the left, rotate it through an angle of three units about some predefined axis, put it in orientation four (inverted perhaps,), and paint it color seven (mauve). For our purposes, such coordinates could be input parameters (arguments) to successively executed logical operations (APL functions), or even indices denoting which functions are to be executed.

This is a nutshell description of Schillinger's Theory of Rhythm. The way he extended it to include "space" in a totally abstract sense, allowed him to apply it as the fundamental theory upon which he could build "theories" of all other components of art forms. In this chapter, I have no intention of being so ambitious. The idea of rhythm that generally comes to mind when one speaks of

"patterns in music" is generally the naive one associated with instruments of percussion. Here, I will *not* take the notion far beyond this jaded level. Remember, I am trying to restrict the discussion so as to apply to the idiom of the musical theater. However, because that idiom deals with words as well as music, there are some surprises in store. When we begin to program rhythmic patterns, our rhythmic naiveté and the shallowness of our subconscious ideas about rhythm become painfully clear. As we become proficient in recognizing rhythms, we will be astounded by the masterfully intricate patterns we use in ordinary speech. It may be the case that our musical culture is evolving rhythmically to make use of the abilities we have already developed in using language. But before getting into this, let's consider Schillinger's treatment of rhythms of duration and the program I promised to show for producing rhythms directly with a computer.

SCHILLINGER'S RESULTANTS OF INTERFERENCE

Schillinger may have overstressed the importance of "interference patterns" in his theory. His algorithm for generating them required pencil, graph paper, and much tedium. In fact, there seems a proportionality between this tedium and the importance he placed on the resultants of interference. One wonders whether he would have given them such weight if he had had a computer and the simple algorithm shown in the APL functions listed in Fig. 7-7.

For example, to find the resultant of interference between the numbers 5 and 3 using his geometric method, you would construct three lengths five units long directly above five lengths of three units and then project downward from each endpoint of all the lengths and measure the resulting intervals, as shown in Fig. 7-8.

To use the computer algorithm directly in APL, you would simply enter on your keyboard,

```
RSLTNT 5 3
```

Almost immediately two lines of output would be displayed:

```
3 2 1 3 1 2 3
2 1 1 1 1 1 1
```

The first line contains the same durations derived by the geometric method (the intervals between endpoints), and the second line shows the number of "generators" simultaneously "attacking" each duration. To appreciate what I mean by tedium, try finding the resultant of three or four numbers—say 9 to 5 to 4—geometrically, and then type **RSLTNT**

```
      ∇ Z←RSLTNT V;ATTK;I;R
[1]     Z←((ρV),×/V)ρI←0
[2]   L1:Z[I;]←(×/V)ρ1,(¯1+V[I])ρ0
[3]     →((ρV)>I←I+1)↑L1
[4]     R←INT(+\∨/[0] Z)⍳1+⍳1++/∨/[0] Z
[5]     Z←R,[¯0.5](0≠ATTK)/ATTK←+/[0] Z
      ∇


      ∇ Z←INT V
[1]     Z←(1↓V)-¯1↓V
      ∇
```

Fig. 7-7. The APL functions RSLTNT and INT compute Schillinger's resultants of interference.

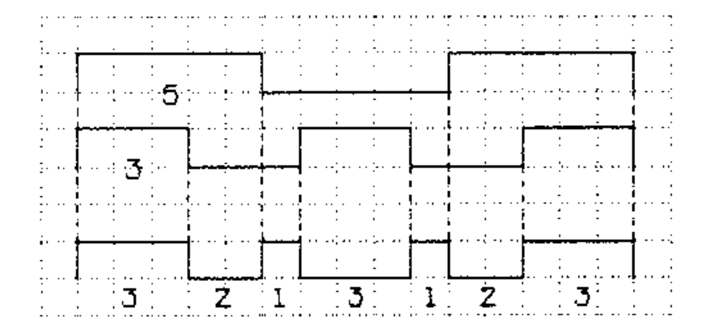


Fig. 7-8. Graphic construction for finding the resultant of interference between five and three.

9 5 4 for comparison. Of course, only if your computer has this algorithm will the comparison run to completion and the 84 durations be fired back at you along with the attack numbers for each duration. When you spend an hour or so geometrically deriving a single rhythm, it is only natural for your results to assume a value commensurate with your efforts. In that same time you might use a computer to print reams of resultants and then find each one as precious as any randomly chosen telephone number from the directory.

Personally, I have found resultants of great value for teaching myself to "hear" and perform them directly as artificial groupings. For example, 9:5:4 would appear in common (4/4) time as shown in Fig. 7-9. But now there is an easier way!

RHYTHMS BY COMPUTER

We will now devise an algorithm for constructing multipart rhythms of any complexity. The results of the algorithm should then serve as input to a computer that can play single notes of any pitch and duration (within "reasonable" bounds). For input to the algorithm, we want to specify a single duration—the time in which each of the rhythms to be specified must fit—followed by any number of attack patterns representing the rhythms. These rhythms can actually be "played" for input by using one hand on one key for the attacks and the other hand on another key for the rests. The idea here is for the user to construct rhythms made up of "regular" parts that can be expressed fairly easily, but which, when played together, form a complex rhythmic pattern.

As a trite example, we would like a dialog of the following sort:

Computer: Enter total time
User: 1 <carriage return>
Computer: Enter rhythm #1
User: X <cr>
Computer: Enter rhythm #2
User: XX <cr>
Computer: Enter rhythm #3
User: XXXX <cr>
Computer: Enter rhythm #4
User: <cr>

By giving the computer a null entry (carriage return only), the user signals that all subrhythms have

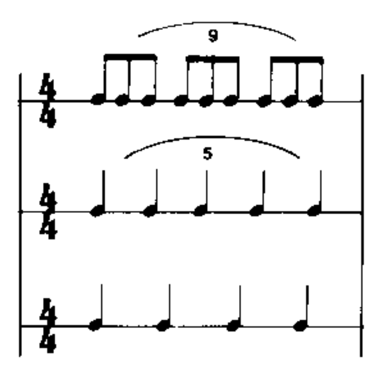


Fig. 7-9. Resultants of interference are useful for learning to "hear" and perform rhythms such as this, where the generators are 9, 5, and 4.

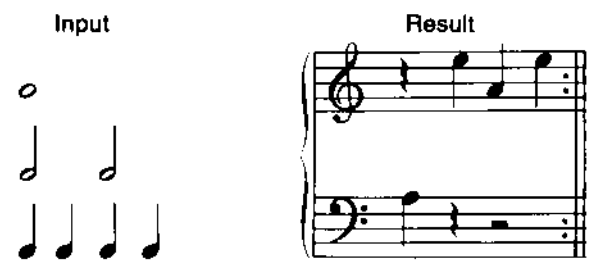


Fig. 7-10. Three inputs (X, XX, and XXXX) would be interpreted by the computer as notated under "Input," and played as shown under "Result."

been stated and the computer is now to play them simultaneously. Each successive subrhythm should sound at a successively higher pitch, and, where more than one attack occurs simultaneously, only the lowest pitch, corresponding to the earliest input rhythm, should be played. To perform the above example, the computer would play the result shown in Fig. 7-10, repeating this continuously until interrupted.

Of course, our purpose is not to work with such common forms. Even the resultants of interference offer little challenge here. For the resultant of five to three, you would enter three X's on one line and five X's on the next. The real rhythmic revolution begins when you ask to hear something like a four-beat background against which a higher pitch is playing as in Fig. 7-11. The background rhythm would be expressed as "XXXX" but the subrhythm in 5/4 time has to be broken down to:

X00000X000X000X000X00X00X00000

That is, each of the five beats must be thought of as assemblages of six units to allow all the attacks to be given precisely. The first quarter note requires an attack (X) followed by five nonattacks (0's) to account for the six units per beat. The quarter-note triplets each occupy one third of two beats or four units (X000). The eighth notes take three units (X00), and the last quarter, like the first, requires a full six units (X00000). (It's easier to "play" this for input than it is to count the keystrokes.) This is the sort of rhythm a composer might "dream up" but wouldn't dare expect anyone to perform—until now!

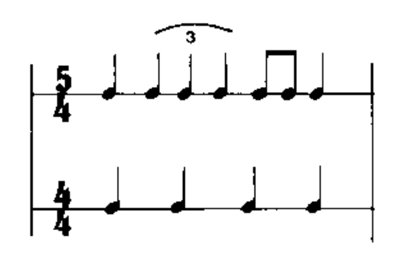


Fig. 7-11. A 5/4 rhythm played against a strict 4/4 background.

The algorithm in APL, up to the point where performance is to begin, could be written as it is listed in Fig. 7-12. The monadic function RHYTHM is to be called with a single alphanumeric character as the right argument. Entering

RHYTHM 'X'

means that rhythms will be expressed as strings of

```
        ∇ RHYTHM ATKSYM;I;K
[1]     F←A←ι0
[2]     'ENTER TOTAL TIME'
[3]     T←⎕
[4]     I←1
[5]    L1:'ENTER RHYTHM ',⍕I
[6]     K←⍞
[7]     →(0∊ρ,K)↑L2
[8]     K←K=ATKSYM
[9]     A←A,K/(ιρ,K)×T÷ρ,K
[10]    F←F,(+/K)ρ110×I×2
[11]    I←I+1
[12]    →L1
[13]   L2:→(0∊ρF)↑0
[14]    K←AιA
[15]    K←(K=ιρ,K)/K
[16]    A←A[K]
[17]    F←F[K]
[18]    K←⍋A
[19]    A←A[K]
[20]    F←F[K]
[21]    D←((1↓A),T)-A
        ∇
```

Fig. 7-12. The RHYTHM function collates separate rhythms into a complex whole.

symbols in which X's specify attacks and any other symbols (including spaces) mark time units of the same duration that are not attacked. The header statement shows the attack symbol will be called **ATKSYM** within the program, and I and K are declared as local variables. The statements in the body of the function serve as listed below.

[1] A frequency vector, F, and an attack vector, A, are initialized as empty vectors.

[2] The user is asked to enter a number representing the total duration in seconds for the rhythms that are to follow.

[3] The user's entry is taken by the variable, T.

[4] A counter, I, is initialized to one, meaning the first rhythm is about to be requested.

[5] The user is asked to enter the Ith rhythm. (Notice, the user is not expected to think in terms of index origin zero!)

[6] The variable named K accepts the literal string entered by the user to represent a rhythm.

[7] If K is found to be the empty vector (shape equals zero), the user has entered only a carriage return signalling there are no more rhythms, so the program is to branch to the statement labeled L2.

[8] The vector, K, containing the user's rhythmic string, is converted to logical form showing attacks with ones and durations that are not attacked with zeroes.

[9] The total duration, T, divided by the number of units in the rhythm (ρ, K) gives the duration for each unit. The instant of time at which each unit duration begins (starting with zero) is found by multiplying the vector 0 1 2 3 . . . by that unit of duration. Compression by the vector K will then leave only the starting times of the units that are attacked. So, for every rhythm defined, the attack vector A has appended to it the times at which attacks occur (relative to a reference time of zero).

[10] Because K now expresses the last subrhythm that was entered as a logical vector, the number of attacks in that rhythm is + /K (the total number of ones). So, for each attack in the rhythm, the Ith harmonic of A below middle C (220 Hz) is here appended to the frequency vector. Notice that A and F will have the same number of entries, and corresponding entries show the time of an attack and the pitch frequency to be played at that time. Incidentally, I had originally chosen a fundamental frequency an octave lower (110 Hz), but, after hearing the tone color of the notes generated by the computer, the factor of two was added. In a more elaborate program, the user could specify a pitch pattern to accompany each subrhythm.

[11, 12] The counter for the rhythms is incremented and we branch back to statement number five (labeled L1) to process the next rhythm.

[13] When statement number seven takes us to this point, we must check to see if any rhythms have been specified. That is, if the user entered only a carriage return for the very first rhythm, the frequency vector will be empty and we must stop execution (branch to zero).

[14] The local variable K is now reset to show, for each attack time in A, the first index of A that contains that attack time. This might be a vector such as 0 1 2 0 4 0 6 2 8, indicating that the first, fourth, and sixth attacks occur simultaneously, and so do the third and eighth attacks.

[15] Repeated entries are removed from K. If K were the above vector, K = ι ρK would be 1 1 1 0 1 0 1 0 1 and this would compress out the repeated values leaving K equal to 0 1 2 4 6 8.

[16, 17] By keeping only the values indexed by K, the repeated entries (representing simultaneously attacked higher pitches) are removed from the attack and frequency vectors.

[18] K is reset to the grade-up of A. This means K will contain the indices of A reordered according to the attack times those indices address.

[19, 20] A and F are reordered to show the actual time sequence.

[21] The difference between the successive entries in A (using T as an appended last value) gives the successive durations between attacks. These values are stored in the duration vector, D.

This program was written upon hearing that APL was available on the IBM Personal Computer and that it could control the sound generator

through an auxiliary processor. The next step was to find out what shared variables are needed for that processor and then write the required function to pass the information contained in the frequency and attack or duration vectors. On gaining access to such a system, however, I found the designers had gone out of their way to be "helpful." Only the most conventional part of conventional notation is implemented (in a coded form, of course) and with the usual failure to account for "artificial" groups. As a result, I could either write my own auxiliary processor or code the whole thing in BASIC. Even if a third alternative, such as having some teeth pulled, were added, writing my own auxiliary processor would come in last. Besides, it seemed (and proved) possible to work out the BASIC code then and there.

The program is listed in Fig. 7-13. It works just as described for the APL program, except that the attack symbol cannot be prespecified (attacks must be given by 1's instead of X's or any other character), a subrhythm cannot contain more than 60 symbols, and a maximum of five subrhythms are allowed.

THE PROBLEM OF SENSIBILITY IN RHYTHM

Earlier I mentioned how useful the resultants of interference have been to me in learning to perform unusual artificial groups. But now I must admit they have had little value in pointing out just what it is that makes a given rhythm appear sensible. The resultants themselves are rarely useful directly as rhythms of durations, even when "spelled out" in a way that does not use artificial groups, because their temporal patterns do not conform to our traditional sense of musical phrasing. They show an overall pattern of durations that reads the same forward and backward! This palindromic symmetry goes unappreciated by our musical sense and leaves unsatisfied our musical needs. Time never flows backwards, and the sound waves produced by musical attacks are neither geometrically nor acoustically symmetric under such time reversal. Just play any tape backwards and you'll see what I mean. So there is no natural reason to expect us to sense such symmetry through hearing.

Another "feature" of the resultants also goes unnoticed by our aural sense. The interference of two numbers creates "families" of duration groups; a resultant contains combinations of duration lengths from one to the smaller generator in successive groups whose sum equals the larger generator. At the same time, successive subgroups add up to the smaller number. In the resultant of five to three, as an example, we see the groups add to five:

(3 2) (1 3 1) (2 3)

and the subgroups contain or add to three:

3 (2 1) 3 (1 2) 3

Such patterns may have great appeal to our analytic sense, but they offer little of interest to senses attuned by evolutionary design to the natural world.

Schillinger was well aware of these properties of resultants, and saw nothing unnatural in them. He felt that beauty recognized by the intellect must have universal appeal to all the senses. His convictions were so firm that he was able to take rhythm as *the* fundamental musical entity and the resultants of interference as an adequate base for rhythm! Though he was a mathematician, he seems to have overlooked the fact that solutions to problems of analysis and synthesis can be quite different. It is possible to analyze a given waveform in terms of combinations of sine waves and then use them to synthesize the sound of that instrument. Similarly, it is conceivable that any given rhythm could be analyzed in terms of combinations of interference patterns, and so the rhythm could be synthesized from them. But if you were asked to synthesize any orchestral sound, without being given a "target" for analysis, you could spend the rest of your days putting together sine waves in endless ways and never find the sound you want. In creating a sensible rhythm, we have the same sort of problem; no predetermined amount of experimental manipula-

```
10 DEFINT I-N
20 DIM RI$(4),L(4)
30 INPUT "Total time: ",T
40 FOR I=0 TO 4
50 INPUT "Rhythm: ",RI$(I)
60 L(I)=LEN(RI$(I))
70 IF L(I)=0 GOTO 90
80 NEXT I
90 N=I
100 IF N=0 THEN END
110 MXL=L(N-1)
120 FOR I=0 TO N-2
130 IF MXL MOD L(I) THEN MXL=L(I)*MXL
140 NEXT I
150 T=18.2*T/MXL
160 DIM K(MXL),F(MXL),D(MXL)
170 FOR I=N TO 1 STEP -1
180 FOR J=1 TO L(I-1)
190 IF (MID$(RI$(I-1),J,1)="1") THEN K((J-1)/L(I-1)*MXL)=I
200 NEXT J
210 NEXT I
220 DIM P(5)
230 P(0)=30000
240 FOR I=1 TO 5
250 P(I)=110*I*2
260 NEXT I
270 I=0
280 JJ=0
290 IF K(I)=0 GOTO 350
300 D(JJ)=T
310 F(JJ)=P(K(I))
320 JJ=JJ+1
330 I=I+1
340 GOTO 440
350 J=1
360 FOR M=I TO MXL-1
370 IF K(M) GOTO 400
380 J=J+1
390 NEXT M
400 F(JJ)=P(0)
410 D(JJ)=J*T
420 JJ=JJ+1
430 I=M
440 IF I<MXL GOTO 290
450 MXL=JJ
460 I=0
470 WHILE I<MXL
480 SOUND F(I),D(I)
490 I=(I+1) MOD MXL
500 WEND
```

Fig. 7-13. The RHYTHM algorithm coded in BASIC for the IBM Personal Computer.

tion of the resultants of interference can create "the" rhythm you want.

To attack the cybernetic problem I must ask, "How do I choose rhythms of duration?" This leads to further questions about the internal "feedback" mechanism that keeps me "on track," accepting certain patterns and rejecting others. Eventually, the problem distills to, "What makes sense rhythmically, and why?" Notice, this form of the question is more objective; it applies to entire cultures. But the answer I'm about to suggest is largely subjective in that it is based on much thought and little in the way of experiment. Melody and rhythm in different cultures appear to have evolved independently, but rhythm and language seem to exhibit a chicken-egg interdependence. The source of our metric sense may not be musical at all!

Speech—even the speech of the most non-musically inclined individuals—carries the imprint of rhythmic sense most plainly. Concentrate on any speaker's rhythm rather than his verbal content to verify this. It would be wise not to choose a boss, parent, or spouse as a test subject. It would be safest to listen to radio or television or to yourself reading aloud. You will notice a tendency to place accented syllables on a remarkably regular beat. Unaccented syllables are usually forced evenly between beats as in simple duple or triple meters. Even during pauses for breath or for finding the proper words, the beat goes on. Exceptions do occur, giving rise to exemplary forms of syncopation and shifts of beat as in mixtures of 2/8 and 3/8 time signatures. The speaker's own rhythmic sophistication and musical experience certainly affect his speech. With quite simple phrases, I can often distinguish between "jazz" and "classical" speakers of English. Even with foreign languages, a discriminating listener may discern hints of phrasing peculiar to the corresponding "folk" music. I have noticed this for Mexican, Brazilian, Greek, and Korean speakers. Also, after the first few weeks in a foreign country, any composer finds himself using the newly familiar "lingual" rhythms in his music.

The idea that words can always be put to music, in any language, misses an important point here. There are no bar lines or formal phrase lengths in a speaker's mind. These are strictly *musical* constructs used to impose larger rhythmic patterns on commonly spoken ones. Without them, a rhythm is not suitably "musical." With them, speech takes on emphasized meanings. The rhythms of speech are extremely complex, yet we "compose" them subconsciously as we string our words together. Musical rhythms are simpler by far, but they impose a consciously intellectual structure on the patterns of language in time, much in the same way that mosaics depicting irregular spatial patterns can be constructed of regularly shaped pebbles and tiles.

The critical factor in composing sensible rhythms then is not the selection of the pebbles—the actual durations that fill a bar—but the devices we use to combine those pebbles into a larger pattern. Simple schemes involving repetitions, permutations, and various kinds of distortions spring immediately to mind for this purpose. Such devices are readily programmed and often lead to satisfying results. But just as often they lead to recognition of what I mean by "our rhythmic naivete."

A simple "nuclear" pattern such as the first bar of Fig. 7-14 cannot be freely permuted if musical sensibility is to be retained. Some fascinating permutations do arise, in the second bar, triplets syncopate nicely over the beat. But many more permutations such as the third bar are so perplexing that just putting them into conventional notation can be a puzzle.

Our methods must include recognition of those "pebbles" that contain more than one duration. The rhythm shown in the fourth measure can be considered to contain these rhythmic kernels:

♪♩♪ , 3 ♫♪ , and ♩

or even, using a "finer grained" analysis,

♪ , ♩ , and 3 ♫♪

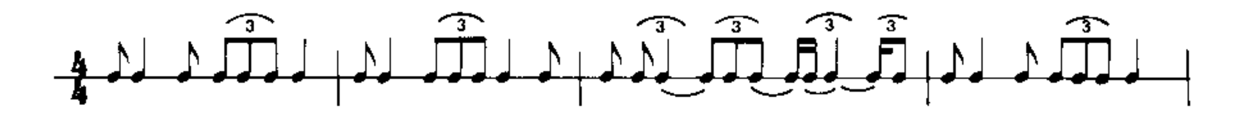


Fig. 7-14. Four measures showing a rhythm, an interesting permutation, a perplexing permutation, and a return to the original rhythm.

We will call such elements *microrhythms* or *micropatterns.* The safest devices for development, i.e., creating extended patterns from these elements, are repetitions and rearrangements of the ordering of the micropatterns. And now, we will go into such devices and methods in some detail.

DEVELOPMENT OF MICRORHYTHMS

Repetition seems a most uninteresting device and, when used with rhythmic elements that have little intrinsic interest to begin with, you might expect it to be synonymous with boredom. That impression grows out of experience with repetitive spatial patterns. Unlike space, musical time carries its own metric marker—"the beat." A repeated pattern doesn't seem repetitious when its relationship to the beat keeps changing. When the duration of a musical measure and the duration of a microrhythm have an unusual ratio, repetition causes a subtle kind of interference to take place, but there is nothing subtle about the interesting effects produced.

Figure 7-15 is an algorithm that will repeat a micropattern, MP, until it fills a time duration, TD. That is, if TD equals five and the micropattern has a duration of three, two more units of duration will be added through repetition. Incidentally, the algorithm "knows" that temporal durations are always positive, so you can use negative entries to indicate rests (of positive duration, of course). For illustration, suppose MP is the duration vector, 2 1 1 1, and TD specifies a total duration of 7. The algorithm then does the following:

[1] The result to be returned, Z, is initialized to the empty vector.

[2] The complete micropattern MP is appended to the entries, if any, in Z. Notice, this statement is labeled because we expect to branch back to it as long as Z doesn't account for the total duration.

[3] If the total duration, TD, is greater than the sum over the absolute values of the elements in Z, branch back to the statement labeled **L1**. In our case, TD equals 7. The first time we reach this point, Z will contain one occurrence of MP whose durations total 5. So back we go to the previous statement, resetting Z to contain a second sequence of the MP values: 2 1 1 1 2 1 1 1 . Now, back at this statement, 7 is compared to 10 (the sum over Z), found *not* to be greater, so the branch is *not* taken.

[4] The local variable, K, is set to a logical vector in which ones will appear corresponding to the durations in Z up to the point where the accumulating total exceeds TD. From there, zeroes will correspond to the remaining Z entries. Notice, **+\|Z** implies **+\2 1 1 1 2 1 1 1**, which equals 2 3 4 5 7 8 9 10. With TD = 7, K becomes the result of

7 ≥ 2 3 4 5 7 8 9 10

which is 1 1 1 1 1 0 0 0.

```
      ∇ Z←TD RHM MP;K
[1]     Z←⍳0
[2]   L1:Z←Z,MP
[3]     →(TD)+/|Z)↑L1
[4]     K←TD≥+\|Z
[5]     Z←K/Z
[6]     Z←Z,(TD)+/|Z)⍴TD-+/|Z
      ∇
```

Fig. 7-15. Function RHM repeats a rhythmic micropattern, MP, to fill a given total duration, TD.

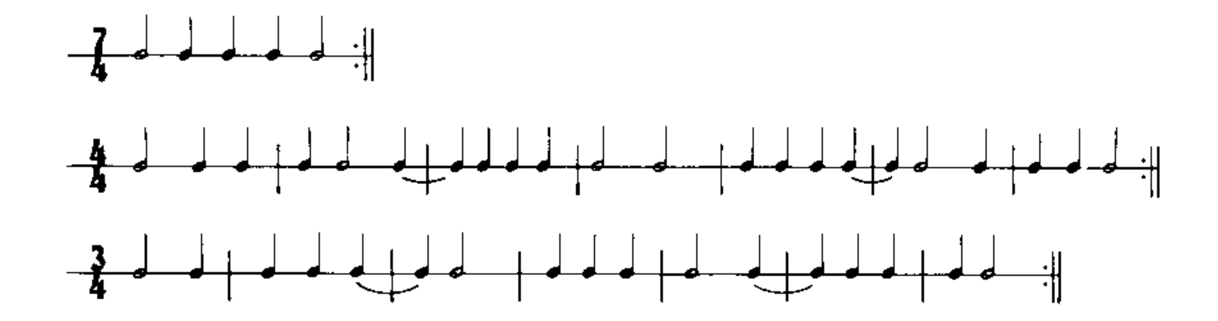


Fig. 7-16. Repeating the rhythm of durations 2 1 1 1 2 in 7/4, 4/4, and 3/4 meters.

[5] Z is now reset by using K to "compress out" the entries that correspond to its zeroes. In our example, K/Z becomes

1 1 1 1 1 0 0 0 / 2 1 1 1 2 1 1 1

which equals 2 1 1 1 2.

[6] An additional value will be catenated to Z if required to fill out the time span specified by TD. This accounts for the possibility that the last duration value used in the repetition will not quite reach the necessary total, but the next duration in the micropattern would exceed that value. For the values we're using here, the expression in parentheses will be zero (the expression, TD is greater than the sum over the absolute values in Z, is false), so only a null element is appended to Z (0 ⍴ 0).

The result of our example will superimpose a feeling of 7/4 time on the meter actually chosen. That is, in Fig. 7-16 we see the motif 2 1 1 1 2 used repetitively in 7/4, 4/4, and 3/4 meters.

Although I described the algorithm in terms of durations, its arguments and result are simply numbers. We could reinterpret TD to represent the total number of attacks to be used in repeating an attack pattern, MP, and producing a result representing the numbers of attacks on sequential beats. Our previous result, 2 1 1 1 2, would then translate from the previous figure to Fig. 7-17.

In fact, using Schillinger's idea that rhythm can be interpreted to apply to any sort of "space," we need not give TD or MP such explicit meanings at all. As numbers, they could even be taken to represent indices to a vector or matrix, so the result would select particular elements from particular arrays. However, because the algorithm was designed with durations in mind, we do have to take into account significant differences between durations and



Fig. 7-17. Repeating the rhythm of attacks 2 1 1 1 2 in 5/4, 4/4, and 3/4 meters.

attacks or indices or whatever else we might use. For example, I mentioned that a negative duration could be interpreted as a rest. I did not say what a negative attack might mean! Unless you can assign such meanings, negative values should be avoided when you want to generate attack patterns.

But let's consider the value zero. A zero duration is meaningless; the algorithm effectively ignores it because it contributes nothing to the sums. A zero attack would correspond to a rest for the beat on which it falls, so you would never include a zero in MP if you are considering durations, but you might if you have attacks in mind. If you want to deal with indices, you would have to account for the index origin. You might redefine the index origin as one, or you could subtract one from all elements of the result! For example, say we have a vector, A, containing the two elements 2 and ¯4. Then

```
A[¯1 + 7 RHM 2 1 1 1]
```

would be A[1 0 0 0 1] or ¯4 2 2 2 ¯4. (Notice, 7 RHM 2 1 1 1 would give the previous result, 2 1 1 1 2, and then subtracting one gives the indices shown.)

To make this example musically meaningful, the vector needs to represent something. Let's consider 2 and ¯4 to indicate successive intervals in a progression of root-tones. That is, beginning with the note C, the roots of the successive chords must move down four semitones, up two, up two more, etc. The APL expression to capture this root-tone sequence could be written:

```
+ \0, A\1 + 7 RHM 2 1 1 1]
```

which would give 0 ¯4 ¯2 0 2 ¯2 or the root-tones C, A♭, B♭, C, D, and B♭. The same "rhythm" or a different one could be applied to select chord structures to be built on these roots.

Using the same rhythm with an array containing just two chords would synchronize root-tone motion with chord structure. For example, if B were the array

```
2 2 ⍴ 3 3 3 4
```

the first row would contain the successive intervals in a minor triad and the second would do the same for a major triad. Symbolically, we could even say B represents the literal array

```
2 3 ⍴ 'MINMAJ'
```

or

```
MIN
MAJ
```

Now, just as we began the root-tone sequence arbitrarily with C, we can specify the first structure to be major. Then the entire sequence

```
MAJ, B[¯ 1 + 7 ¯ 2 1 1 1;]
```

taken together with the root-tones derived above would be:

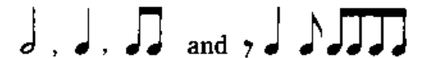
C MAJ, A♭ MIN, B♭ MAJ, C MAJ, D MAJ, B♭ MIN.

STYLE

When you do things in a consistent way, you have a style, but when you do them consistently and with taste, you have style. Every composer has a style because everyone's knowledge is limited and that imposes consistency through constraints on ideas. The goal of all composers is to outgrow their particular (limited) style and develop style—the genuine article without the indefinite article. But no matter how we try to phrase it, the facts will seem rather cold to anyone with romantic notions. Perceived gradations of style are purely subjective. As a composer, you outgrow your style only when you recognize the limitations of your methods and increase your knowledge so as to develop more sophisticated methods. A description of methods for attaining "style" will prove far more enlightening than the most carefully worded generalities.

Even an inexperienced listener will recognize, or at least sense, style based on repetitions and permutations of rhythmic micropatterns that are not too sophisticated. The presence of the same kernels throughout a rhythm provides consistency, while their changed order supplies "taste." Sophistication depends on the micropatterns themselves as well as on the particular devices for reordering them. To some extent, sophistication, complexity, and style become inseparable.

As an example, suppose we limit ourselves to these four microrhythms:



The first three of these are totally lacking in "style" because of their brevity and "respect for the beat." The last one is not so colorless. Now, to give substance to the example, I'm going to specify an attack rhythm of 6 4 6 3. These will be the numbers of attacks in four successive measures. Each set of attacks must be constructed from the given microrhythms. In choosing two sixes, I have in mind the use of the fourth microrhythm for one of them and some combination of the "styleless" ones for the other. The other two numbers come to mind because there are four microrhythms and three of them have no "character" of their own. These numbers simply "come to mind" for the reasons given; in reality, they are more arbitrary than my "reasons" would indicate. Equally arbitrary is the way I now assign duration to the attacks. The first four measures might appear as shown in Fig. 7-18. Notice, the listener will not be aware of the three simplest microrhythms. Instead, the more specialized motifs that fill the individual measures will stand out as "the" rhythmic micropatterns. Further development must take this into account before introducing any new combinations of the basic microrhythms.

Repetition is necessary to fix these patterns in the listeners' minds. But direct repetition of the above will produce the most banal sort of eight-bar phrase, so I purposely will show an "insertion" technique. We'll repeat the first two measures, insert the first and fourth bars next, and then repeat the third and fourth measures of the original; the result is shown in Fig. 7-19.

The entire sequence of ten bars will suggest all sorts of further development to a composer. A programmer can continue by relying on the simplest numerical methods. For example, the ten attack values now are

6 4 6 3 6 4 6 3 6 3.

The first thing to strike me (and purely at a visual rather than intellectual level) is the prevalence and regularity of those sixes. Next, I feel the value four has somehow been cheated; just looking at the sequence of non-sixes, I expected the last number to be a four. So with all the whimsy at my command, I'll tack on four sixes and a final four. The last five bars might then be the ones shown in Fig. 7-20.

Combining this with harmony suggested by the previous example, in which root-tones and chord structures were synchronized, melodizing by methods to be taken up in due course, and extending the whole—rhythm, melody, and harmony—into more complete temporal form (also to be discussed), we have a result (Fig. 7-21) that does indeed have a style, if not style.

Rhythmically, though perhaps not harmonically, I have stayed well within the limits of style for the MT idiom here. For typical meters and

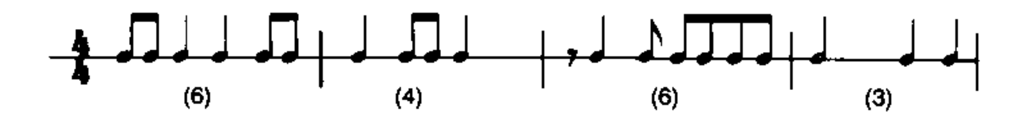


Fig. 7-18. Four measures based on the attack pattern 6 4 6 3 and the four microrhythms shown in the text.



Fig. 7-19. The rhythm of Fig. 7-18 with an "extra copy" of the first and fourth measures inserted between the second and third measures of the original. (The sequence of measures becomes 1 2 1 4 3 4.)

tempos in that idiom, any beat or half-beat may be attacked to produce a "sensible" micropattern. Within that simple constraint, psychologically acceptable microrythms have quite free form except for one further consideration. As a rule of thumb, it would be wise to base development on repetition of a few short phrases, ensuring that the listener gets the point in our musical conversation, and preventing saturation of his or her mind with too many different patterns. A reasonable guideline might restrict the micropatterns to lengths not exceeding one or two measures. For the thoughtful composer, I should now prove that, as limiting as these rules may seem, the MT style has not been completely exhausted.

Consider a single measure of four beats, eight half beats. In APL we can construct a delightfully simple representation using logical vectors (ones and zeros).

1 0 1 0 1 0 1 0 =

1 1 1 0 1 0 1 1 =

0 1 0 1 1 1 1 1 =

With eight entries in a vector, the ones indicate which of the eight potential attacks actually are to occur. Zeroes will not necessarily indicate half-beat rests; they simply show the absence of an attack. (Coincidentally, one such measure corresponds to one byte, and the computerists among us might enjoy thinking of rhythms as ASCII characters or two hex digits.)

The combinatorial mathematics of this situation (two things taken eight at a time) indicates that there are 256 possible one-measure rhythms (including the four-beat rest, i.e., no attacks in the measure). Pondering the number of bars of 4/4 music already in existence, in which all attacks (if any) occur on an eighth note, will lead one to the inescapable conclusion that there are no new microrhythms to be found within the restrictions given above. However, an even smaller number of elements accounts for all things in the physical universe. So we must take into account the number of "diatomic musical molecules" that can be formed by selecting any two one-measure rhythms to be played consecutively. This turns out to be the same as the number of values addressable in two bytes—the well-known "64K" or 65,536 two-measure rhythms.

Taking the argument one or two steps farther, we find there are over 16 million rhythms three bars in length and more than 4 billion four-measure rhythms. If you could find and play each of those *four-plus* billions of measures in real time at a tempo of 120 (beats per minute), you would finish in about



Fig. 7-20. Five measures developed from the attack rhythm 6 6 6 6 4.
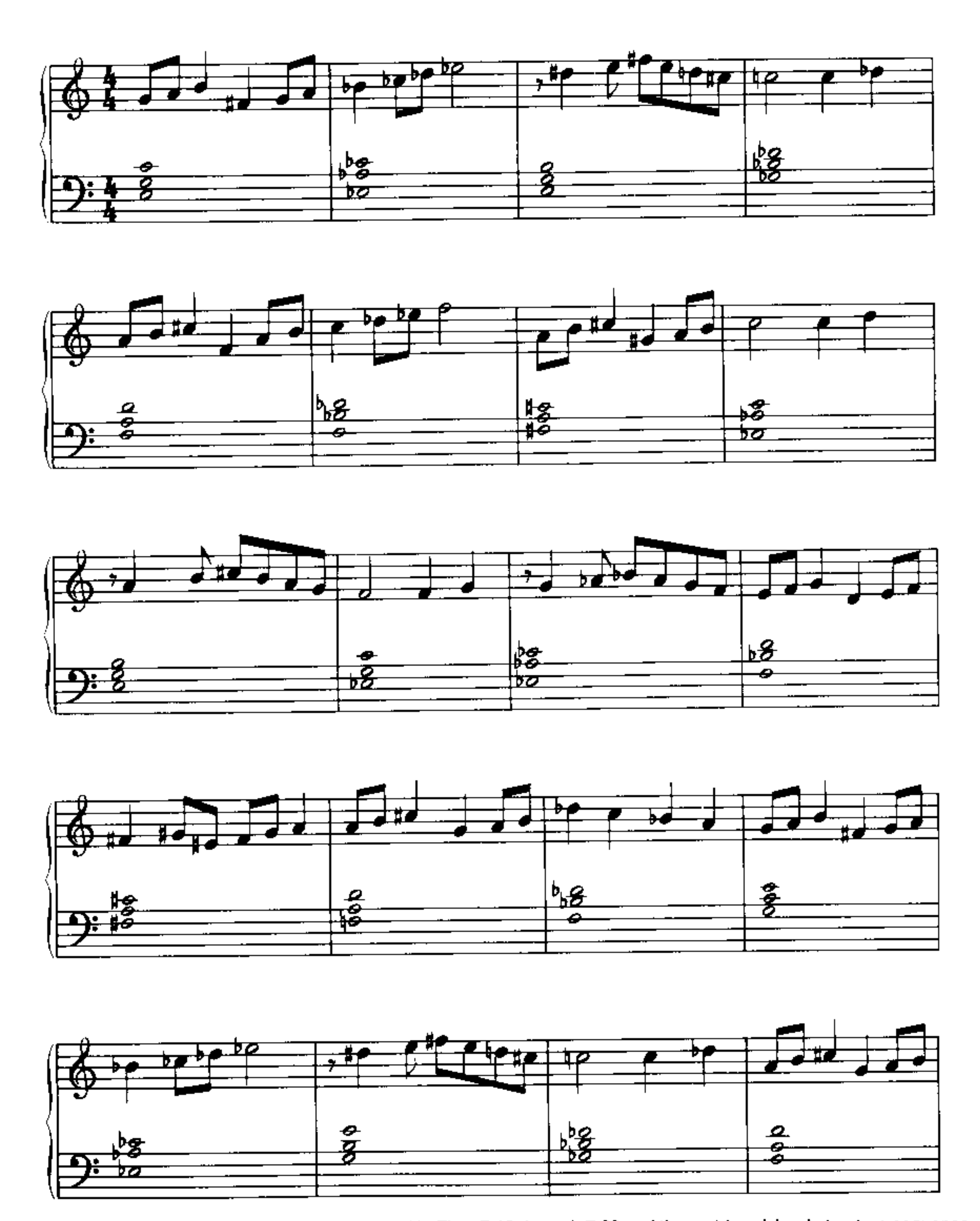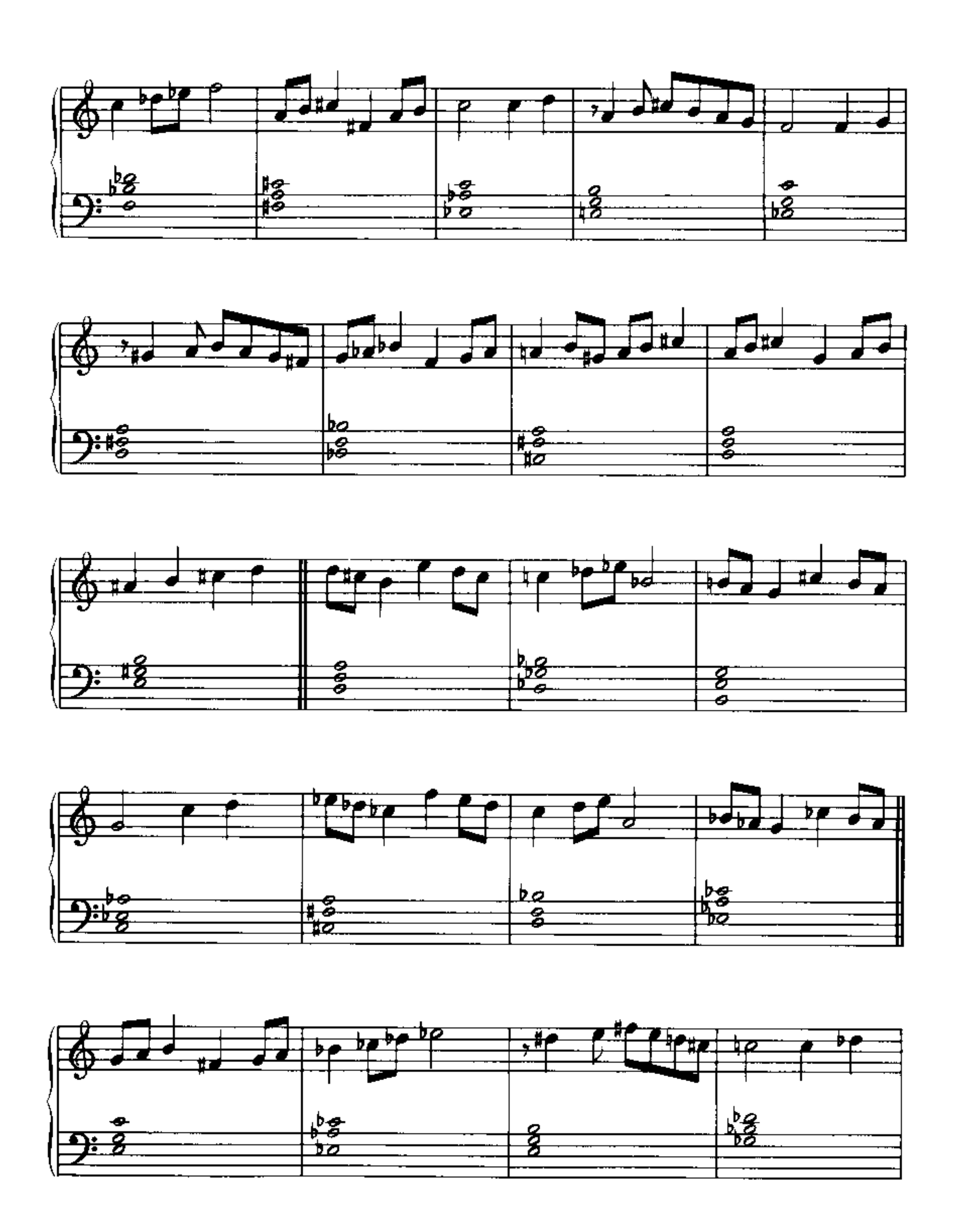


Fig. 7-21. A piece based on the rhythm developed in Figs. 7-18 through 7-20 and the root-tone/chord-structure sequence developed on page 63 (©1978 Jaxitron).
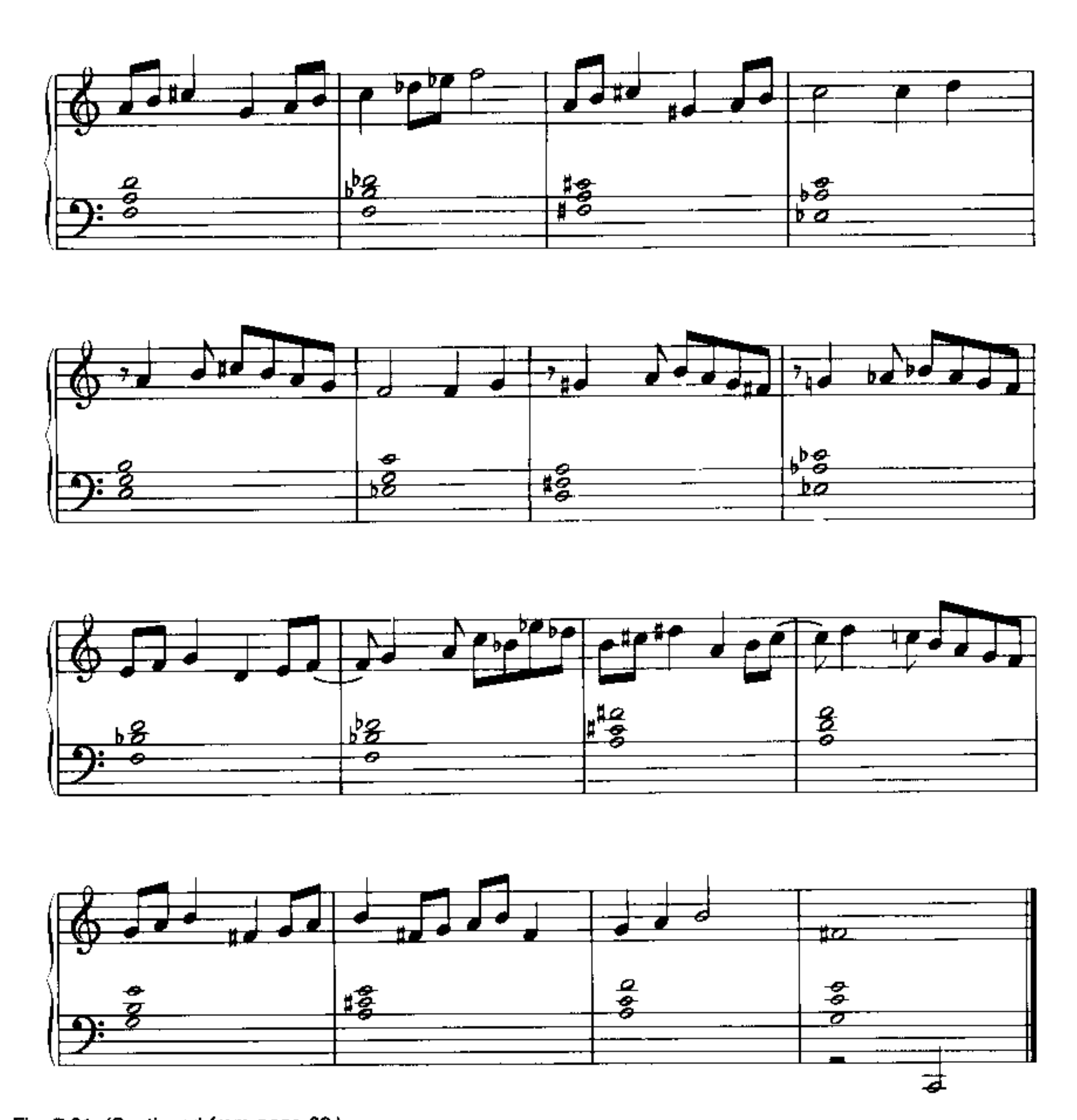




Fig. 7-21. (Continued from page 66.)

A.D. 3000! The MT idiom as we know it has been around less than a tenth of that time, so we needn't be surprised when new tunes seem to keep appearing endlessly. As I will try to make clear later, the astronomical numbers that appear when simple elements are combined are the direct cause of our need for repetition of a small number of patterns. Such numbers also account for "sensibility" in a way that is itself quite sensible and yet most unexpected.

PERMUTATIONS

Repetition can also be made interesting by changing the order of the repeated elements, even if the total pattern remains "in sync" with the meter. Techniques for reordering (or permuting) such

elements are easily programmed in APL and provide informative examples. Here are a half-dozen illustrative techniques for permuting the elements of a vector. In each case, the vector will be represented by the symbol V. We will also use a variable, P, whose numerical form will depend on the specific permutation technique under consideration.

Permutation by Index

Here, P will be the desired order of the indices. With V = 'ABCD' and P = 3 2 0 1, our result would be **V[P]** or **'DCAB'**. Notice that rows or columns of arrays can be permuted in the same way by writing

```
RESULT ← ARRAY[P;]
```

or possibly even

```
RESULT ← ARRAY[V[P];]
```

(where V is a numeric vector). Now, suppose V is the attack vector 2 3 4 1 representing the rhythm shown in the first bar of Fig. 7-22. **V[P]** then gives **(2 3 4 1)[3 2 0 1]** = 1 4 2 3, the second measure. Repeating the process on this result: **(V[P])[P]** = 3 2 1 4, produces the third measure. One more time, **((V[P])[P])[P]** = 4 1 3 2, creates the last.

If you were to apply this permutation again to the last result

```
4 1 3 2 [3 2 0 1],
```

the original vector, 2 3 4 1, would reappear. So this particular permutation of four different objects (P = 3 2 0 1) can produce just four different results by "recycling" the operation. In our example, four one-beat microrhythms are ordered to form four micropatterns, each one bar long. We can call these V, V1, V2, and V3, with the number indicating the number of times the permutation was carried out. That is,

V2 = (V[P])[P]

where the "2" implies two P's. Notice V4 = V, V5 = V1, V6 = V2, etc.

Now we can use the same technique to extend the four unique measures into four different four-measure phrases! By reordering the sequence (V, V1, V2, V3) with the same permutation vector, (3 2 0 1), we produce the four measures shown in Fig. 7-23. And, by calling the original four-bar sequence, S, and the above permutation, S1, we realize that S2 and S3 can be derived from them just as V2 and V3 were derived from V and V1. So, although this permutation "only" rearranges four things in four different ways, it also rearranges that sequence of four arrangements in four different ways. You can reread the previous sentence until either you fall asleep or see that an endless sequence can be constructed which *never* actually repeats (though after listening to a few hundred bars it may seem to be repeating endlessly).

Permutation by Value

To establish a particular ordering, P, for the elements in V, we could write

```
RESULT ← V[PρV]
```

However, in defining P, perhaps interactively, it would be too easy to include invalid entries. Sup-

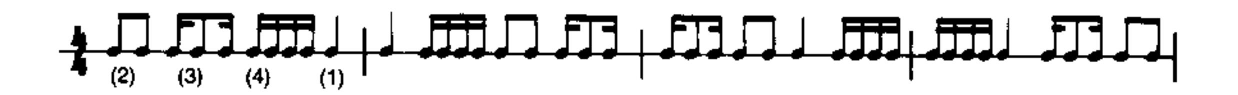


Fig. 7-22. The 2 3 4 1 attack pattern (first measure) is shown as single, double, and triple 3 2 0 1 permutations by index in the succeeding three measures.

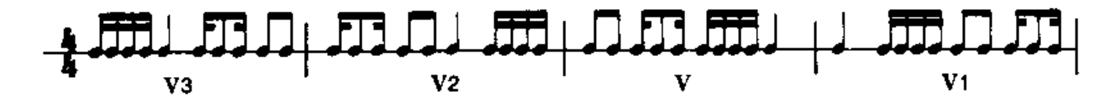


Fig. 7-23. The same vector used in Fig. 7-22 (3 2 0 1) used to permute the four permutations!

pose V equals 'DACB' and somehow I manage to ask for a permutation, P, specifically to be 'AEBD'. Then, just to be safe, the programmed statement using P could be

```
RESULT ← V, '*') [P⍳V]
```

RESULT would then turn out to be the vector, 'A*BD', and instead of stopping with an "index error" message, the program could continue with a preplanned evasive maneuver.

For a numeric example, consider the attack vector used in Example 1. We might want the sequence 1 4 3 2, which was not attainable by the permutation used there. Obviously, we could simply reset V to that sequence. But in a complex situation, where we need to be sure we're are not altering the raw material, a better form would be

```
RESULT ← (V, ¯1,) [P⍳V]
```

followed by

```
→ (¯1∊RESULT) ↑ ERROR
```

where ERROR labels a statement containing our instructions for handling any emergencies caused by invalid entries in P.

Cyclic Permutations

The rotation function in APL performs cyclic permutations directly. The assignment

```
RESULT ← P⌽V
```

causes the Pth element of V to become the first (zero) element of RESULT. If V were 'ABCDE', the following table shows the effect of various values of P.

P	RESULT
0	'ABCDE'
1	'BCDEA'
3	'DEABC'
¯1	'EABCD'
¯4	'BCDEA'

It is as if the elements of V form a continuous loop with A following E directly. Calling the first element's index zero lets us refer to other indices as either positive or negative integers in a perfectly logical way. Thus, E is both *element* number four as well as *number* negative one.

Notice that any cyclic permutation can be "recycled." If P and the number of elements in V do not have a common divisor, each possible cyclic arrangement of V will occur before recycling returns the original order of the elements. With P equal to 2 and five elements in V, the first rotation (V1) would be 'CDEAB'. Rotating that with the same P produces V2 = 'EABCD', and so on until V5 is found to be the same as the original V.

Reflection

Remember, the same operator used for rotation performs the reversal function when applied monadically:

⌽ 'ABCDE' produces 'EDCBA'

This operation, usually called a *reflection* by musicians, cannot be recycled without returning the original setting immediately; a reflected reflection appears the same as no reflection!

Pair Interchange

The reversal operator can be used in a more complex expression to cause pairs of entries in a

vector to trade positions. Suppose V equals 'ABCDEF' and P is set to 0 2 1 5, meaning that the elements in positions zero and two are to be interchanged, as are those in positions one and five, producing the result, 'CFADEB'. To do this formally in APL, we would write

```
RESULT←V
RESULT[P] ←,⌽(((⍴P) ÷ 2), 2)⍴ V[P]
```

As was true for reflection, recycling here returns the original order of the vector. But you might see what happens if you choose different pairs of P vectors and use them alternately. You might also try permuting the elements of a single P vector!

Generalized Reflection

This permutation can be called a *rotated reflection* if you remember that a rotation implies a cyclic shift of components and a reflection simply reverses the order of elements. In explaining this operation, the idea of a two-sided mirror reflecting the elements of the vector is useful if not accurate. The position of the mirror determines the "axis" of reflection. But following this reversal, the mirror suddenly acts more like a window in that the number of elements seen through either side remains unchanged. That is, placing a mirror midway across the B:

'ABCDE' would ordinarily produce

'EDCBA'

but then the "window effect" changes this so that

'ABCDE' produces

'CBAED'

Placing the "mirror" midway between the D and E

'ABCDE' happens to produce the same

'CBAED'

The reason for explaining the operation in such a contrived way is to allow a scalar value to determine the result. As for the cyclic shift, positive or negative integers will address elements in the vector for positioning the mirror. But half-integers can also be used. A value of one for P would correspond to the first case in which the mirror cut across the B. Negative four would do the same. The second case could be characterized by P values of 3.5 or ¯1.5. Notice, the values .5 and ¯.5 are equivalent to the ordinary reflection. An APL expression to represent this operation can be written:

```
RESULT←(-(⍴V) |1 + 2×P) ⌽⌽ V
```

A RECURSIVE GENERAL PERMUTATION ALGORITHM

At times a particular permutation scheme may not suggest itself, so the composer may think he wishes to see all possible permutations of his material. Figure 7-24 lists an algorithm that will serve in principle. I say "in principle" because "all possible permutations" may easily overtax the available resources. This algorithm is directly useful when

```
      ∇ Z←PERM V;I;J;K
[1]     Z←(0,⍴V)⍴I←⎕IO
[2]     K←UNIQ V
[3]     →(2=⍴V)↑L2
[4]   L1:J←((V⍳K[I])≠⍳⍴V)/⍳⍴V
[5]     Z←Z,[⎕IO] K[I],PERM V[J]
[6]     →((⎕IO+⍴K)>I←I+1)↑L1
[7]     →0
[8]   L2:Z←((⍴K),2)⍴V,⌽V
      ∇

      ∇ Z←UNIQ V
[1]     Z←((⍳⍴V)∊V⍳V)/V←,V
      ∇
```

Fig. 7-24. Function PERM captures all permutations in the vector V.

the number of unique elements in the vector to be permuted is relatively small, say smaller than six or seven. Six different items can be permuted 720 ways; seven items have 5040 permutations. But the length of the vector also must be considered because all its elements may not be unique. That is, **'ABC'** will have six permutations of the three elements with a total of 18 characters to be displayed in the list of results. But the same unique elements in the vector **'AAABBCCC'** will have (8!) ÷ (3!) × (3!) × 2 or 560 permutations requiring a list of 4480 characters. If all eight elements in this vector were different, there would be over 40,000 permutations and nearly a third of a million characters to be displayed.

The second statement in the function **PERM** calls the function UNIQ which returns a vector containing just the unique elements of the original vector. But notice the function that is called in the fifth statement, **PERM**! A function that calls itself is called a *recursive* function. Such behavior in anyone deserves closer scrutiny, so let's examine just what happens here. Because recursion is a rather advanced topic, if the level of detail becomes oppressive, feel free to skip to the next section.

The initial step of the algorithm, **[1]**, sets the variable I as a scalar zero (our index origin) and also establishes array Z to be returned when execution is completed. This array is initialized here with the same number of columns as there are entries in the vector to be permuted, but with no rows, so it begins as an empty array. Next, in **[2]**, the local variable K is introduced so as to isolate just the unique elements of the original vector V. Now, a test is made in **[3]** to determine whether V has just two elements. For the moment, let's suppose it does. The algorithm then branches to the last statement, **[8]**, labeled L2. Here the array to be returned, Z, is shaped to have as many rows as there are unique elements in V. Because, with two elements, V has either the form "aa" or "ab," Z will be returned as the one-by-two array "aa" in the first case, or as the two-by-two

```
a b
b a
```

in the second.

Let's return to statement **[3]** now to see what happens when V has more than two elements. The branch to **[8]** is not taken, so statement **[4]** is executed next. The local vector J is set up to contain all the indices of V other than the one that contains the first appearance of the unique element, **K[I]**. But, to be specific, we'll say V has three components and the index I equals zero. This makes **K[I]** the same as the first entry in V. And now we can look closely at the first recursive call to **PERM** in statement [5].

Reading from right to left, we see **PERM** is called with **V[J]** as the vector to be permuted. This vector must be **V[1 2]** because, as we just saw, J will not contain the first index of **K[0]** in V, which must be zero. We know that when **PERM** operates on a two-element vector, there are two possible results: "aa" or the two-row matrix containing "ab" and "ba." Statement **[5]** then prefixes the returned result with **K[I]**, which we recognize as **V[0]**. So just this much of the statement is equivalent to an array that depends on the form of V. If the original three-element vector had the form "aaa," then **K[0]** catenated to **PERM V[1 2]** would return the unsurprising result, "aaa." With V = "abb," we could expect the equally unsurprising result "abb" because the first element, "a," will be catenated to the only "permutation" of "bb." The remaining possibilities for V are "aab," "aba," and "abc." For each of these, **PERM V[1 2]** returns a two-by-two array, so with **K[0]** = "a" the possibilities are:

```
a a b    a b a    a b c
a b a    a a b    a c b
```

To complete the statement, the particular result of **K[0],PERM V[1 2]** is then catenated row-wise to the previously set array, Z, which is empty when I = 0.

In the next statement, **[6]**, I is increased by one and tested to see if there are more unique elements to be taken into account. For the case of V = "aaa," K has only a single entry so the test fails and statement **[7]** ends execution. For any other form of V, K will contain at least two elements. So with I now set to one, the branch is taken, and we find

ourselves back at statement [4]. K[1] is now the second unique entry in V, so that J is either 0 1 or 0 2. The possible results for the recursive step can now be tabulated;

(I = 1)	J	V	K[I],PERM V[J]
	0 2	abb	bab bba
	0 1	aab	baa
	0 2	aba	baa
	0 2	abc	bac bca

The appropriate result is then appended to the previous one and again statement [6] increments I and tests it against the number of unique elements. Only then K has a third element will we branch back one more time. The remaining entries to be appended to the previous ones must then correspond to I = 2 and J = 0 1, so that, with V having the form "abc," K[I],PERM V[J] is

cab
cba

Instead of returning to statement [3] and considering the chain of events when V has four entries, five entries, and so on, let's back off a bit and observe the whole procedure with some perspective. No matter what the length of the original vector, the PERM algorithm sets aside one unique element and then calls PERM to treat the remaining elements.

The effect of recursion and the reason for using it should now be evident. This process of peeling off unique elements continues with the successive calls until a level is reached with only two elements. After permuting them, the algorithm returns to the call at the three-element level, and continues as we just discussed until all permutations of these three are in place. Retrogression continues in this way for the last four elements, the last five, and so on until completion. Of course, the sequence of events for four elements is more convoluted than it was for three, but there is no difference in the logic. In fact, when you analyze a logical situation for N elements and find that you can describe it in a way that depends on N – 1 elements, your analysis need go no further; you have a recursive algorithm.

FAMILIES OF RHYTHMS

We now have the ability to see all permutations of a particular rhythm. A more fundamental and useful ability would show us all motifs that serve the same rhythmic function. Such a collection comprises a family of rhythms, all of which have the same number of attacks and fill the same total duration. But, having seen that a single rhythm can have an astronomical number of permutations (and each permutation is certainly in the same family), we had better limit this list to all "unpermuted" members of the family. An unpermuted rhythm will be defined as one in which the durations are represented by an ordered string of integers with no retrogression of value. All unpermuted rhythms having two elements and a duration of five would then be 1 4 and 2 3. These could be considered the "seed stock" for the entire family, and their "offspring," 4 1 and 3 2, are generated by permutation. For three elements that add up to five, we can expect the seeds 1 1 3 and 1 2 2. Our permutation algorithm can give us the rest of the family.

I have named the functions of an algorithm that find the seeds of a family BREAK and SPR8. A composer with a particular rhythm in mind can ask for

N BREAK T

or

FAMILY ← N BREAK T

where N is the number of attacks in his rhythm and T is the total duration. It is usually easier to find N and T than to express a complex rhythm numerically. These functions are listed in Fig. 7-25.

```
      ∇ Z←SPR8 V;X
[1]     Z←(0,ρV)ρ0
[2]     X←V[⍋V]
[3]     →(2=ρV)↓L2
[4]     L1:Z←Z,[⎕IO] X
[5]     X←X+ 1 ¯1
[6]     →((1↑X)≤1↓X)↑L1
[7]     →0
[8]     L2:Z←Z,[⎕IO] X[⎕IO],SPR8 1↓X
[9]     X←(1+¯1↓X),(¯1↑X)−¯1+ρX
[10]    →((1↑X)≤¯1↑X)↑L2
      ∇

      ∇ Z←N BREAK T
[1]     Z←SPR8((N−1)ρ1),T−N−1
      ∇
```

Fig. 7-25. Functions BREAK and SPR8 collect all "unpermuted" rhythms with N attacks and a total duration of T units.

If you enjoyed the analysis of the recursive function in the last section, you'll love figuring this one out yourself. If you didn't, fear not! I will just make a few comments here to clarify how to approach such a problem.

For most people, it helps to transcribe problems of this sort into terms more easily visualized. In this case, consider each attack of a given rhythm to be replaced by a cup containing pennies, where the number of pennies in a cup equals the duration of the corresponding attack. The number of attacks then becomes the number of cups, and the total duration becomes the total number of pennies in all the cups. Our first task might seem to be to rearrange the cups to be sure that no cup has more pennies than the one to its right or fewer than the one to its left. This would represent the seed for the rhythm. But we need to generate all the seeds from one seed, and this particular rhythm, even when "unpermuted," may not give us the family's primary progenitor.

So we need to visualize two limiting distributions of pennies, and a procedure for converting one distribution into the other, a penny at a time if necessary, so that when we reach the final distribution we will have passed through all the seed distributions. The algorithm begins by placing a single penny in each of the cups except the last; that last cup gets all the remaining pennies. The other limiting distribution, the one we will try to reach by moving one penny at a time, is the one that comes closest to an equal distribution of the pennies (without violating the rule that no cup can contain more pennies than the one to its right).

We begin by working with just the last two cups. We take one penny from the rightmost cup and place it in the cup to its left; equivalently, we subtract one from the last duration and add one to the penultimate one. Next, we must test to see that all values (durations or pennies per cup) are still in nondecreasing order. If the test proves positive, we place this rhythm in our list and repeat the operation, subtracting one from the last duration and adding one to the next-to-last. Sooner or later, the last two cups will reach their limit of equality and the test must fail. We then discard the improper rhythm and put all the pennies back in the original configuration—one in each cup except the last, which gets all the remaining pennies.

We now take two from the last cup and place one in each of the two preceding cups. If this distribution passes the test to avoid decreasing values, we accept it as the next rhythm and also as a "second-generation seed" for further rhythmic derivation. This new seed marks the starting point for subjecting the last three cups to the one-penny treatment. When those three reach their limit of equality the test fails again, we return to the original seed and invest three cents in the production of a new second-generation seed which will affect only the last four cups. The seemingly unending nature of this logic shows the unmistakable mark of a recursive process.

RHYTHMIC SELECTION

For rhythm, as for the other components of music, the problem of selection differs distinctly from that of generation or derivation. We have shown that a computer can derive enormous numbers and

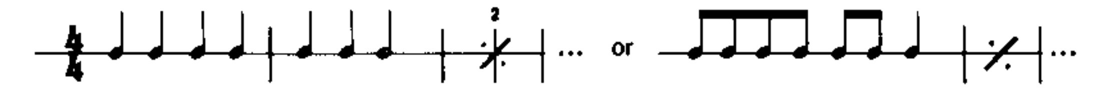


Fig. 7-26. A most uninspiring rhythm for a composition.

varieties of rhythms to serve a composer's needs. The algorithms we used were models of mechanical schemes for moving numbers around as if they were strung on a necklace, reflected in mirrors, or passed through sieves for grading and caught in containers. Unfortunately, the mechanics of any selection process are far more complex.

Rhythm is rarely an independent component of music in the MT idiom. It bends to the "strain" of melody and even to the pressures of harmony. Moreover, this idiom features language in a way that forces rhythm to imitate the role it plays in ordinary speech. The composer must portray the written word with *musical* rhythms the way a painter depicts visible textures with brush strokes. Pitch and time are his pigment and canvas. As an artist, the composer selects material by searching through his technical resources for an approach that will satisfy his medium, his idiom, and his style. Only when that process of searching can be modeled in a way that produces realistic results can his "art" move to a higher level of "search and select." The problem of selecting rhythms by computer is nearly as complex as that of having a computer extemporaneously construct intelligent speech. Neither problem is likely to be solved immediately.

To demonstrate the intricacies of the problem, I'll describe my subjective mental processes on being given lyrics that are to be set to music. Many of the examples to be shown throughout this book came into being through collaboration with lyricists in classes and workshops for musical theater writers. Of course, most of those collaborators were unaware of my devious methods of composition. But my experience leads me to view the fit between lyrics and rhythm to be of three classifiable types.

Dum-de-dum-de-dum-de-DUMB

The least inspiring and most difficult to deal with is the 4/4 trochee, a string of alternately accented and unaccented syllables in common meter, typified by William Blakes's well-known "Tiger, tiger burning bright." (See Fig. 7-26.) Today, even children are so influenced by the rich musical styles that serve as themes and background to their video favorites that their musical IQs are too high to take such rhythms seriously. Yet there are times when an interesting lyric falls naturally into this damnable category. I recall one in particular that had the "natural" meter shown in Fig. 7-27, repeated to fill three 16-bar sections of a particular form for a song. One solution would be to use this rhythm and rely on the melody and harmony to keep the listener awake. But that's not really a solution; it just shifts the problem to another musical component and may prove even harder to solve.

Years ago, a stylistic solution was applied which amounted to giving the stressed syllables a larger share of the beat either through the use of triple meter (the first line of Fig. 7-28) or through "shuffled" eighth notes in performance (the second line).

Such a slight deviation from "regularity" helps until it, too, becomes commonplace. Taking a cue from this, my solution was to alter the length of



Fig. 7-27. An uninteresting rhythm suggested by an interesting lyric.



Fig. 7-28. An outdated way to add interest to the rhythm of Fig. 7-27 (first line), as well as an alternate form creating the same effect (second line).

every other unaccented syllable (Fig. 7-29). Reaction was interestingly mixed. Lyricists with little training but with obvious "natural" talent for music found it thoroughly enjoyable. Musicians were put off by my cumbersome time signature, 3 ½/4 (7/8). Because the tempo was quite bright (about 240 quarter notes per minute), most (even the musicians!) thought I was playing in a strict duple meter, not recognizing the beats they counted as I played had a duration ratio of four to three. A most perfect display of our rhythmic naivete! With repeated hearing, all strangeness here vanishes and this sort of metric manipulation becomes quite automatic. I frequently find myself now applying this solution even to waltz meter, changing 3/4 to 2 ½/4 (5/8).

Easy Rhythm

The second category of lyrical rhythm is a joyful relief from "problems" of any sort. The words have a natural quality, free of forced or overly obvious patterns of stressed and unstressed syllables, yet readily grouped into phrases that meet musical requirements. Some writers have an uncanny ability to compose such lyrics, whether producing words alone or fitting them to a given melody. An example comes to mind in which the words immediately established the rhythm. The noteworthy point, in musical jargon, was that a five-measure "bridge" fell out as naturally as eight-measure A-sections in its three AABA format. (If that makes no sense, don't worry about it. We'll clear it up when we discuss form.) An added bonus with this type of lyric is the gratification one experiences when the lyricist's talent is misinterpreted as the composer's ingenuity.

Lyrically Rhythmic Lyrics

Even more enjoyable, though problematical from the composer's standpoint, is the third type of lyric, in which clear musical phrasing exists but the microstructure is intriguingly irregular. A classic example would be Longfellow's "The Day is Done" (Fig. 7-30). While each stanza might be forced to fit the same eight-bar phrase, there are numerous places where single syllables would be smeared over multiple pitches or single pitches would need to accommodate multiple syllables. More serious still, there doesn't appear to be any single setting that avoids rhythmic awkwardness in parts of each stanza. So, each requires its own treatment, and yet needs to be as integrated



Fig. 7-29. A new solution to the rhythmic problem posed in Fig. 7-27.
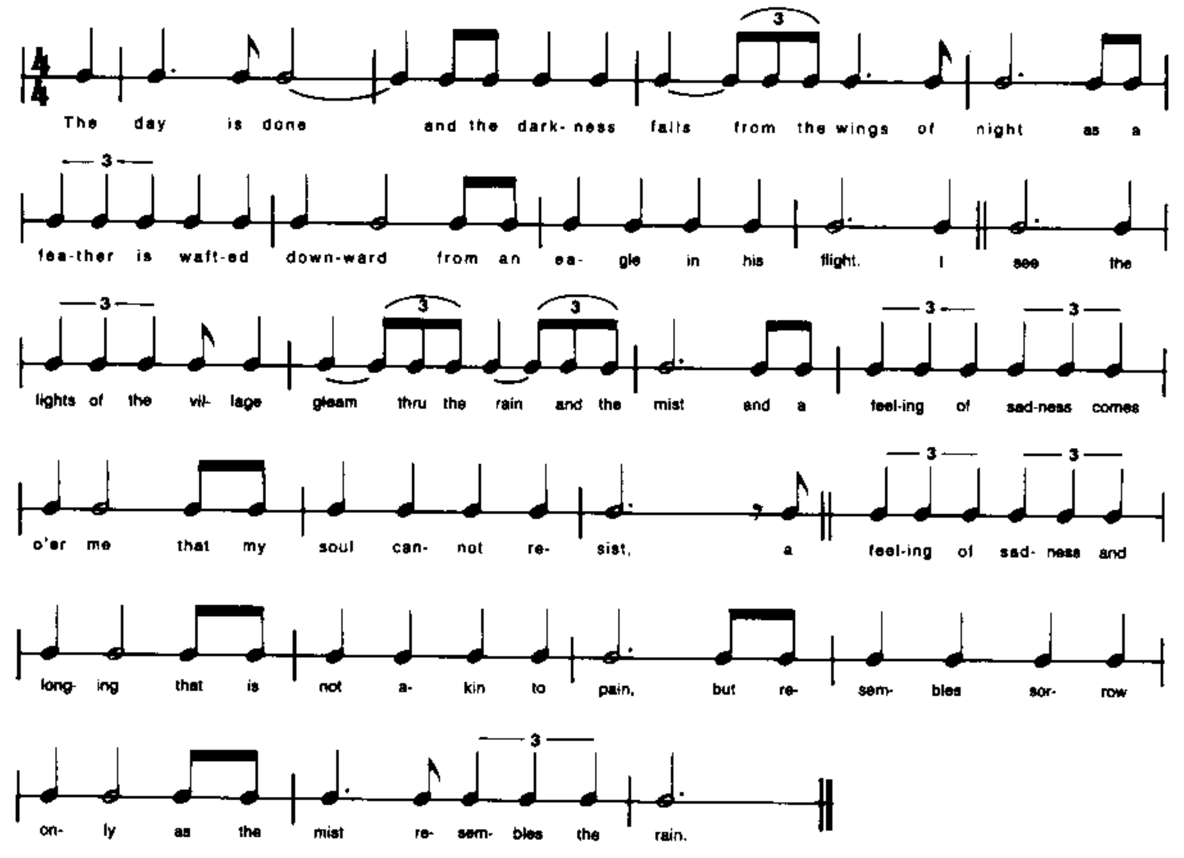


Fig. 7-30. A rhythmic setting for Longfellow's poem, "The Day is Done" (© 1978 Jaxitron).

musically as it is poetically. Here is an example of a setting for the first three stanzas. Each is treated as similarly as seems possible while avoiding undue stress on "unimportant" words or rushing through significant ones. The test for practicality is to see if the words make sense when spoken in this rhythm, possibly with a bit of rubato or anticipation where such interpretation seems fitting.

8 Raw Material

The nature of rhythm is such that we can see a clear difference between techniques for generating raw rhythmic material and for selecting motifs to be used in a finished product. With the other musical components, selection and generation are not always so distinct. It would be stretching a point to say that one *selects* a method of *generating* a pattern, yet for melody and harmony, such a pattern is often useful directly as thematic material no matter how abstract its origin. In fact, we will see later that techniques for selection of those components can indeed be modeled on purely numeric methods, so that whether they are selected or generated becomes a moot question.

In this chapter I want to show methods that can be considered generating techniques. "Generating" is clearly the right word because there will be no musical goal in mind. In the past it was not uncommon for a patient and careful musician to derive tables of numbers, convert them into musical notation, and sell them as finger exercises, instrumental studies, and catalogs of scales, melodic patterns, harmonic structures, and harmonic progressions. Today, a few lines written in APL can produce volumes of such material. A few lines minimizes the need for patience and care, and volumes of material places supply so far ahead of demand that the possibility of selling the results is also minimal. Nevertheless, such programs will provide you with unimaginable wealth which you can spend as you choose in your own compositions.

PITCH SCALES BASED ON INTERVALS

For the moment, let's ignore the psychological importance of the octave in musical scales. We can construct a scale by defining a sequence of intervals that does not necessarily "come out even" with the octave. Put another way, a freely chosen interval sequence might generate pitches that show an "interference pattern" with the octave. If we choose the interval sequence 2 2 1, and specify middle C as a reference point, in the vicinity of middle C the scale would appear as shown in Fig. 8-1. The function SCLI (Fig. 8-2) will do this for us in terms of pitch numbers rather than pitch names.

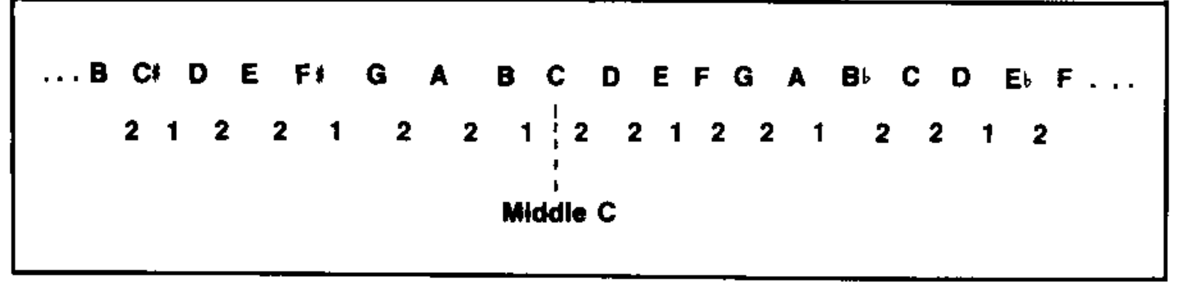


Fig. 8-1. The interval pattern 2 2 1 (referenced to middle C) generates an "interference pattern" with the octave.

We can call **SCLI** with a right argument consisting of any interval sequence whose sum does not equal zero. For example,

SCLI 2 2 1

will construct a scale centered on zero (assumed to represent middle C) in which the given intervals repeat so as to fill a range from five octaves below zero to five octaves above. The scale, consisting of numbers rather than pitch names, will be stored in the global variable named SCL. Because this is such a long numeric vector (spanning ten octaves!), the function explicitly returns the index of the value zero (middle C) in the vector. The function makes use of a predefined global variable named TONSYS. This is a scalar declaring the number of tones per octave in the tuning system. For all our work in the MT idiom, TONSYS is set to 12.

The effort required to read a list of ten-octave scales is not much less than that needed to sing them. Figure 8-3 is another short function devised to limit the range of the scale to be displayed. Now, typing

¯1.5 SHOWSCL 3.33333

will show us that part of **SCL** within the range from one and one-half (¯1.5) octaves below zero to three and one-third (3.33333) octaves above.

CONVERTING NUMBERS TO PITCHES

Before going on, one further convenience must be introduced because, as sure as computers can compute, numbers can numb. Figure 8-4 shows the functions and global variables I used to translate from numbers to pitch names in the examples to be shown.

N2P is mnemonic for converting Numbers to Pitches. The right argument used in calling this function can be any list of pitch numbers. That list will be converted to a similarly shaped array of pitch names. Included in the output will be certain numeric indicators for clarifying the octave position of the notes. (We'll see exactly how this works shortly.) The function **N2NP** was cloned directly from **N2P** so as to produce the same results without any octave information. (Note: Though

```
      ∇ Z←SCLI I;J
[1]     J←((ρI)×⌈(5×TONSYS)÷+/I)ρI
[2]     SCL←(⌽+\0,-⌽J),1↓+\0,J
[3]     Z←SCL⍳0
      ∇
```

Fig. 8-2. Function SCLI creates a ten-octave scale based on the sequences of intervals in the vector I.

```
      ∇ MN SHOWSCL MX
[1]     MN←MN×TONSYS
[2]     MX←MX×TONSYS
[3]     ((SCL≥MN)∧SCL≤MX)/SCL
      ∇
```

Fig. 8-3. Function SHOWSCL displays part of a scale within a given range.

```
      ∇ KN←N2P A;N;O;KD;KF;FLDS;ORHO;I
[1]     →(0=ρ,A)/0
[2]     N←(¯2↑1,1,ρA)ρA
[3]     KN←TONSYS|N
[4]     KD←⌊N÷TONSYS
[5]     O←((¯1↓ρN),1)↑KD
[6]     KD←KD-(O, 0 ¯1 ↓KD)
[7]     KF←(KF×(|KF)=(⌊/|KF←KN∘.-NUM)∘.×(ρNUM)ρ1),TONSYS
[8]     KF←(1=+\+\KF≠0)×KF
[9]     KF←(¯1↓ρKF)ρ(,KF≠0)/,KF
[10]    KF←(KF≠TONSYS)×KF
[11]    KN←NTS[NUM⍳KN-KF]
[12]    FLDS←4+⌈/,|KF
[13]    ORHO←¯1↑ρKN
[14]    KN←((¯1↓ρKN),ORHO×FLDS)ρ(,KN),((ρ,KN),¯1+FLDS)ρ' '
[15]    N2POUT 1
      ∇


      ∇ KN←N2NP A;N;O;KD;KF,FLDS;ORHO;I
[1]     →(0=ρ,A)/0
[2]     N←(¯2↑1,1,ρA)ρA
[3]     KN←TONSYS|N
[4]     KF←(KF×(|KF)=(⌊/|KF←KN∘.-NUM)∘.×(ρNUM)ρ1),TONSYS
[5]     KF←(1=+\+\KF≠0)×KF
[6]     KF←(¯1↓ρKF)ρ(,KF≠0)/,KF
[7]     KF←(KF≠TONSYS)×KF
[8]     KN←NTS[NUM⍳KN-KF]
[9]     ⍝ FLDS←4+⌈/,|KF...REPLACE NEXT STATEMENT WITH THIS
[10]    ⍝ IF TONSYS IS NOT EQUAL TO 12
[11]    FLDS←5
[12]    ORHO←¯1↑ρKN
[13]    KN←((¯1↓ρKN),ORHO×FLDS)ρ(,KN),((ρ,KN),¯1+FLDS)ρ' '
[14]    N2POUT 0
      ∇


      ∇ N2POUT Z
[1]     I←0
[2]     →(L1,L2)[FLDS=4]
[3]     L1:KN[;I+1+FLDS×⍳ORHO]←(1 0 1 \FORS)[(KF=0)+2×KF≥1]
[4]     KF←(×KF)×(KF≠0)×¯1+|KF
[5]     →(L2,L1)[(FLDS-4)>I←I+1]
[6]     L2:KN←' ',KN
[7]     →Z↓0
[8]     KN[;FLDS×⍳¯1↑ρKD]←(1↓DGTS)[|KD]
```

Fig. 8-4. Functions and variables for translating numbers to pitch names.

```
[9]   KN[;¯1+FLDS×1↓ι¯1↑ρKD]←('¯ +')[1+× 0 1 ↓KD]
[10]  KN←(⍕O),':',(' ',KN)
     ∇
        TONSYS
 12

        NUM
0 2 4 5 7 9 11

        FORS
↓↑

        NTS
CDEFGAB
```

reference to certain local variables was removed from **N2P** to form **N2NP**, I neglected removing them from the header.) **N2NP** stands for Numbers to Nominal Pitches. These are the only two names a user need be concerned with. Both functions call **N2POUT** to display the results, and all three functions rely on the global variables TONSYS, NUM, FORS, and NTS being in the form shown. NUM contains the numeric equivalents of the pitch names in NTS. Flats or sharps are indicated by the two arrow characters in the variable FORS. (Early in the game, the ♯ and ♭ were not available.)

SCALES BASED ON INTERVALS IN A FIXED RANGE

Recall how the function named **RHM** repeated a microrhythm to fill a specified total duration. A similar algorithm can restrict an interval sequence so that it just fills one octave. Repetition of that octave will then produce a scale that sounds more conventional simply because the "same" notes appear in every octave. However, letting convention guide rather than guard our methods, we might choose any range to limit the repetition, not necessarily an octave. That is, suppose we had a function that would repeat an interval sequence, I, until it filled a total interval, T:

I BRK T

We could then use that bounded sequence as the right argument of **SCLI** to repeat the limited sequence over ten octaves:

SCLI I BRK T

If T were 12 and I contained 2 2 1, **BRK** would construct 2 2 1 2 2 1 2, and **SCLI** would use that for creating the ten-octave scale. But it would be easier for a user to remember and to write the form:

T SCLIT I

to do the same thing. The function, **SCLIT**, wouldn't really do all this; it would simply use **BRK** and **SCLI** as indicated above to carry out the process. The necessary functions are listed in Fig. 8-5.

To state the case mnemonically: To build a scale (SCL) based only on an interval sequence (I), use **SCLI** with the sequence of intervals as the (right) argument. To construct SCL based on an interval sequence and a total limiting interval (T), add "T" to the name of the function and include the total interval as a left argument:

<u>T</u> SCLI<u>T</u> I

CATALOGING SCALES

Noting that I and T need not be commensurable, T need not equal the number of pitches in an octave of the tuning system, and the number of pitches per octave (TONSYS) need not equal twelve, you may wonder how many scales can be created. To generate different scales using **SCLI**,

```
      ∇ Z←T SCLIT I;J
[1]     J←I BRK T
[2]     Z←SCLI J
      ∇


      ∇ Z←V BRK N;K;R;J
[1]     K←+/J←(0≠V)/,V
[2]     Z←((⌊N÷K)×ρ,V)ρV
[3]     R←K|N
[4]     L0:R←R-+/K←((+\J)≤R)/J
[5]     Z←Z,K
[6]     →(R=0)/0
[7]     →((R=J[0])∨R=1)/L1
[8]     →(R>J[0])/L0
[9]     →(0<ρJ←1↓J)/L0
[10]    R←Rρ1
[11]    L1:Z←Z,R
      ∇
```

Fig. 8-5. Functions SCLIT and BRK build pitch scales based on the interval sequence in vector I that repeat over a total interval, T.

different interval vectors are needed. Conventional scales usually contain only the intervals one and two, so we might consider how many interval vectors can be created using just these values.

If the vector can have only one entry,there are two possibilities, 1 or 2. Allowing two intervals, we see four different vectors:

1 1, 1 2, 2 1, 2 2.

Similarly, three elements admit eight vectors:

1 1 1, 1 1 2, 1 2 1, 1 2 2, 2 1 1, 2 1 2, 2 2 1, 2 2 2.

In general, vectors with N elements can be set up in 2*N (two to the Nth power) ways. Even if we account for the fact that a vector of one length may produce the same results as some shorter vector, there will be an abundance of unique results remaining. And now, if we forget about convention and permit any pair of intervals (1 and 3, 2 and 3, 1 and 4, 2 and 4, 3 and 4, . . .), each pair will generate as many results as the original 1 and 2.

Accepting any three values for the scale intervals

(1 2 3, 1 2 4, 1 3 4, 1 2 5, 1 3 5, 1 4 5, 2 4 5, . . .)

increases the variety to 3*N sequences with N entries for each triplet of values. And for N elements allowing M different values, each M-tet will have M*N different vectors. The need for some sort of bookkeeping should be apparent here. Knowing that two intervals can give rise to 128 different seven-interval scales is not of much value without a way to keep track of those scales. And so the function **ELEMENTS**, listed in Fig. 8-6.

Now the seven-number vectors generated from the integers two and five can be called forth using:

```
7 ELEMENTS 2 5
```

This will cause all the combinations to be computed. A different vector of twos and/or fives will be set in the variable Z each time the function executes the statement labeled L1. The very next line, beginning with the "lamp" symbol, can be replaced with any statements you choose to convert the Z-vector into a scale and display it. For example, replacing that line with

```
R ← SCLI Z
```

uses each Z as a kernel for constructing a ten-octave scale. Inserting directly after this the statement:

```
¯2 SHOWSCL 2
```

will display the four central octaves of each scale. Actually, only the pitch numbers would be shown. To catalog the pitch names , the inserted statement could be changed to

```
N2P ¯2 SHOWSCL 2
```

or

```
      ∇ N ELEMENTS I;D;GD;RGD;Z
[1]     RGD←ρGD←I[⍋I]
[2]     D←0
[3]    L1:Z←GD[(NρRGD)⊤D]
[4]    ⍝INSERT STATEMENTS USING Z HERE.
[5]     →((RGD*N)>D←D+1)↑L1
      ∇
```

Fig. 8-6. Function ELEMENTS generates all number strings of length N containing combinations of the numbers in I. Additional statements may be inserted in place of the one beginning with the "lamp" symbol to treat the strings as desired.

N2NP ¯2 SHOWSCL 2

or even

(⍕Z), ':', N2P ¯2 SHOWSCL 2

The last example would prefix to each scale the Z-vector that generated it.

CONVOLUTED AND SYMMETRIC SCALES

SCLI and SCLIT do not produce different sets of scales. An informed user could devise an interval sequence for SCLI and a related sequence plus a "T" interval for SCLIT that would cause both functions to produce the same scale. Then why have both available? With two algorithms, the composer can use different approaches to the problem of creating scales. SCLI lets him think in terms of single sets of intervals. By adding one *control parameter*, SCLIT makes it easier for him to show style in his use of scales—that is, to create related patterns in different scales. One sequence of intervals with different total subintervals (T) will create a "family" of such scales. The same T value coupled with different interval sequences will produce different scales in which only the interval of repetition is the same. If that interval is not an octave, subtle effects of a stylistic nature can be achieved.

But there is no reason to limit ourselves to just two methods. In fact, after working with them for a while, at least two situations will arise that suggest the need for another form of stylistic control. Both situations arise from patterns of intervals that turn back on themselves.

Starting with middle C, the sequence 2 ¯1 ¯2 2 will produce
C D C♯ B C♯ E♭ D C D E E♭ C♯ E♭ F . . .

Embedding this pattern within a ten-octave "scale" can confuse the composer as well as his listeners. On viewing the middle range of such a scale, it is not immediately apparent whether the basic sequence (the stylistic kernel) begins on the C shown above, on the next C seven pitches later, or on the previous one (not shown) which would be two pitches earlier.

The other situation occurs when one chooses a pattern that happens to end just where it began. This means the intervals add to zero, and so cannot be used at all. The technical reason for banning such a sequence lies in statement [1] in SCLI were division by +/I will cause a "domain error" message. The logical reason stems from the idea of repeating the sequence of intervals until it fills a larger interval.

The algorithm listed in Fig. 8-7 consisting of the functions SYMSCL and FRCTN provides another approach to the formation of "scales" and also solves both the preceding problems. FRCTN merely serves to reduce the fraction

(TONSYS | T) ÷ TONSYS

to lowest terms. SYMSCL will produce the pattern of intervals (I) beginning on zero. The pattern will be repeated T units away from middle C and repeated again another T units away. These repetitions continue until the pattern would again begin

```
      ∇ Z←T SYMSCL I;J
[1]     Z←+\0,(¯1+⌈/TONSYS FRCTN(TONSYS|T)÷TONSYS)⍴T
[2]     Z←,+\Z,((⍴Z),⍴I)⍴I
      ∇


      ∇ Z←M FRCTN F;A;I
[1]     I←1
[2]     L1:A←(I⍴0),I÷(I−1)↓1+⍳M
[3]     Z←A⍳F
[4]     →(Z≤M)/L2
[5]     →(M≥I←I+1)/L1
[6]     Z←'?'
[7]     →0
[8]     L2:
[9]     Z←I,Z
      ∇
```

Fig. 8-7. Functions SYMSCL and FRCTN create "symmetric" scales.

on C (any multiple of TONYSYS away from zero). If T is set equal to the number of pitches in an octave, only one cycle of the pattern will appear. Any other value for T will produce one of Schillinger's "symmetric scales"—hence the name **SYMSCL**. That is, for T equal to 3, the basic interval sequence will begin on C and be repeated successively beginning on E♭, F♯, and A. T values of ⁻3, 9, or 15 would give the same results except that the starting notes of the successive subcycles of the pattern would be permuted, undergo octave transposition, or both (Fig. 8-8 and 8-9).

Here, T equals four, so repetitions occur starting on E and A♭ (G↑). Notice also in this first example showing use of the function **N2P**, the pitch names displayed are preceded by "**0:**" indicating that the following pitch lies in the octave that starts on middle C. Subsequent pitches also lie in this same octave until another number appears. The fourth pitch (A) is preceded by "⁻1," indicating that the following note(s) are in the next octave below the previous one. The appearance of "+1" two pitches later takes us back to the middle octave where we stay until another such numeric indicator appears.

FROM SCALES TO CHORDS

Any traditional seven-note scale contains traditional triads—three-note chords built in thirds above a "root" or root tone (Fig. 8-10).

```
      4 SYMSCL 7 ¯2 ¯8 2 7 ¯5 2
0 7 5 ¯3 ¯1 6 1 3 4 11 9 1 3 10 5 7 8 15 13 5 7 14 9 11

      N2P 4 SYMSCL 7 ¯2 ¯8 2 7 ¯5 2
0:  C    G    F  ¯1A    B  +1F↑   C↑   D↑   E    B    A    C↑
 D↑   A↑   F    G    G↑ +1D↑   C↑ ¯1F    G  +1D  ¯1A    B
```

Fig. 8-8. Using function SYMSCL directly and as an argument of N2P.
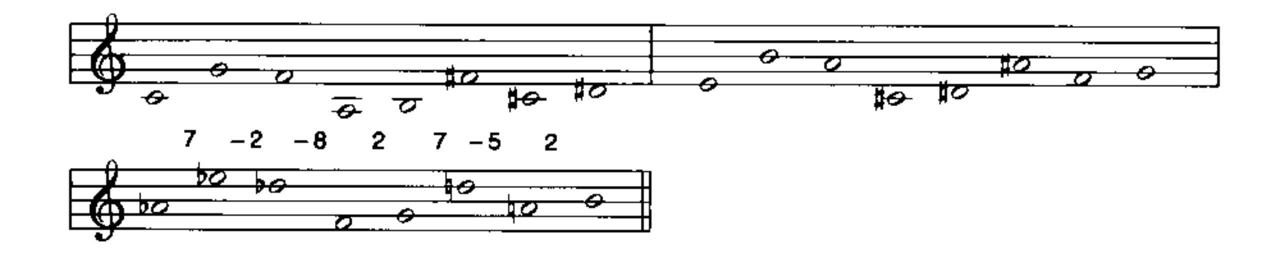


Fig. 8-9. Musical result showing the "scale" produced in Fig. 8-8.

The algorithm that is stored in a musican's mind and lets him extract chords from a scale goes as follows. The notes of the scale are numbered from one to seven. A triad built on the root whose position in the scale is one will contain one, three, and five of the scale. Similarly, if the root is two, the chordal notes are two, four, and six of the scale. An expression in APL that will show all the triads in such a scale can be written:

SCALE[7|(ι7) ∘.+2×ι3]

This would create a list of seven rows and three columns showing the pitch numbers that make up the triads built successively on the pitches in SCALE. For a C-major scale, SCALE would be 0 2 4 5 7 9 11. The above expression would use the seven sets of scale indices shown in Fig. 8-11 on the left to produce the array of pitch numbers shown in the center, implying the notes on the right.

Having captured the traditional logic, we can restate it in a more general form and see where it leads. The simplest way to do this is to consider all numbers in the expression to be *parameters*. (A parameter is a constant value in one situation, a different value in another.)

SCALE[N|(ιN) ∘.+E×ιS]

With N equal to 7, E equal to 2, and S equal to 3, this expression is identical to the previous one. But now N is generalized to mean the number of tones in the scale and need not be restricted to one octave, E is an expansion factor (E equals 2 implies successive chordal tones are two scale tones apart), and S is taken as the number of notes in a chord
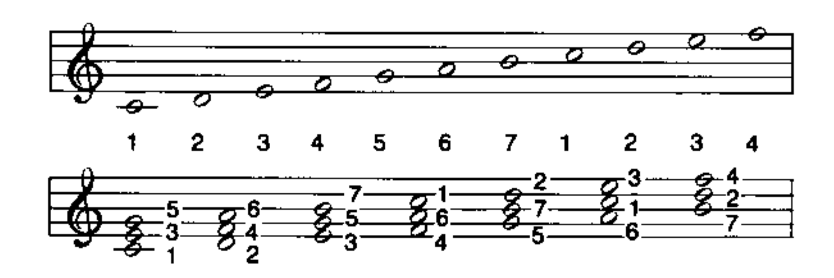


Fig. 8-10. The conventional triads contained in a C-major scale.

SCALE Indices	Array	Notes
0 2 4	0 4 7	C E G
1 3 5	2 5 9	D F A
2 4 6	4 7 11	E G B
3 5 0	5 9 0	F A C
4 6 1	7 11 2	G B D
5 0 2	9 0 4	A C E
6 1 3	11 2 5	B D F

Fig. 8-11. Seven sets of scale indices produce pitch numbers, which in turn imply the pitch names shown in the right column.

structure. Now, each setting of N, E, and S will provide a list of structures, and each setting may be applied to any scale we might create.

To clarify the effect of E on the C-major scale, still keeping S equal to 3 and n equal to 7:

E = 1 produce the "triads" (C D E), (D E F), (E F G), . . .
E = 2 establishes (C E G), (D F A), etc. as already shown
E = 3 implies (C F B), (D G C), (E A D), . . .
E = 4 causes (C G D), (D A E), (E B F), . . .

and so on. Of course, all this can be, and has been, put in functional form (Fig. 8-12).

Here, **XPND** accepts as arguments a scale on the left and a two-element vector containing E and S values on the right. The number of entries in the scale establishes the value of N. In the example shown, **SCLIT** creates a ten-octave scale with middle C having an index of 45. Then **SCL[45 + ι9]** extracts a nine-tone scale for the left argument. Right arguments of (2 4), (3 4), and (4 4) produce four-note chords in expansions of two, three, and four. Notice that when E equals three, the first and fourth notes of each chord are the same. This would not happen for seven-note scales, seven being a prime number while nine is divisible by three.

CHORDS BASED ON INTERVALS

Like scales, chords can be derived directly from chromatic intervals. Figure 8-13 is a function named after Schillinger's "sigma structures" for doing just that. Typing **SIGMAS 3 4** causes this program to derive all chords containing only unique pitches separated by intervals of three and/or four semitones. As if too lazy to print the entire catalog of such structures, the program asks the user to state the size of the chords he wishes to see. By entering the vector, 4, 9, we are requesting just enough exertion to show us all the four-note and nine-note structures that contain the given intervals (see Listing 8-1 at the end of the chapter).

Though only on the fringe of tradition, serial or twelve-tone music antedates computers and so cannot be blamed on them. Hence, I see little danger in a short demonstration. After modifying statement **[15]** to ensure that a line of output would fit on a standard page, I learned there are four twelve-tone structures containing only major and minor thirds. Serious serialists should not read anything occult into the fact that all four happen to be palindromic—the interval sequences read the same forwards and backwards. For structures limited to three, five, and/or six semitones, I found 92 twelve-tone constructs. Sadly, this could not be programmed to fit on one page, and it didn't seem worth the effort to try outwitting a printer that refused to avoid the perforations between pages (Listing 8-2).

Another way to assemble harmonic structures from intervals relies on what Schillinger called "harmonic strata." As the term implies, layers of small structures are superimposed to form a total structure in this technique. One specifies the number of strata, the intervals allowed in each stratum, and the permitted intervals betwen strata. Listing 8-3 is a catalog of 324 structures resulting from the case where the lowest stratum has the three possible forms shown in the first line of Fig. 8-14. The middle stratum (the second line) has three more, and the upper stratum (the third line) has six. The intervals separating (adjacent tones in) the lower two strata are either six, seven, or eight semitones, and the separation for the upper pair of strata is either four or five semitones.

Following the notes of each structure shown in

```
     ∇ Z←SCL XPND ES;N
[1]    N←⍴SCL
[2]    E←ES[0]
[3]    S←ES[1]
[4]    Z←SCL[N|(⍳N)∘.+E×⍳S]
[5]    Z←(⍕Z),(((1↑⍴Z),4)⍴(' :  ')),N2NP Z
     ∇


            12 SCLIT 1 2 1 1 2
  45


       SCL[45+⍳9] XPND 2 4
 0  3  5  8 :   C    D↑   F    G↑
 1  4  7 10 :   C↑   E    G    A↑
 3  5  8 11 :   D↑   F    G↑   B
 4  7 10  0 :   E    G    A↑   C
 5  8 11  1 :   F    G↑   B    C↑
 7 10  0  3 :   G    A↑   C    D↑
 8 11  1  4 :   G↑   B    C↑   E
10  0  3  5 :   A↑   C    D↑   F
11  1  4  7 :   B    C↑   E    G


       SCL[45+⍳9] XPND 3 4
 0  4  8  0 :   C    E    G↑   C
 1  5 10  1 :   C↑   F    A↑   C↑
 3  7 11  3 :   D↑   G    B    D↑
 4  8  0  4 :   E    G↑   C    E
 5 10  1  5 :   F    A↑   C↑   F
 7 11  3  7 :   G    B    D↑   G
 8  0  4  8 :   G↑   C    E    G↑
10  1  5 10 :   A↑   C↑   F    A↑
11  3  7 11 :   B    D↑   G    B


       SCL[45+⍳9] XPND 4 4
 0  5 11  4 :   C    F    B    E
 1  7  0  5 :   C↑   G    C    F
 3  8  1  7 :   D↑   G↑   C↑   G
 4 10  3  8 :   E    A↑   D↑   G↑
 5 11  4 10 :   F    B    E    A↑
 7  0  5 11 :   G    C    F    B
 8  1  7  0 :   G↑   C↑   G    C
10  3  8  1 :   A↑   D↑   G↑   C↑
11  4 10  3 :   B    E    A↑   D↑
```

Fig. 8-12. The XPND function and an example that creates a scale using SCLIT and then extracts four-note chords in expansions of 2, 3, and 4.

```
    ∇ SIGMAS I;K;A;J;S;Z
      'ENTER VECTOR OF SIGMAS TO BE LISTED'
      S←⎕
      '          INTERVALS:   ',⍕I
      J←2
      A←⍉ 0 1 ⍴I
    L1:Z←(2 1 0 ⍉(1,(1↑⍴A),⍴I)⍴I),((⍴I),⍴A)⍴A
      Z←((×/2↑⍴Z),¯1↑⍴Z)⍴Z
      K←(TONSYS×K=0)+K←TONSYS|+\1,Z
      Z←(∧/[0] 1≥+/(1+\TONSYS)∘.=K)/[0] Z
      →(~J∊S)/L2
      60⍴'*'
      →(0=⍴Z)/0
      (⍕1↑⍴Z),' SIGMA ',(⍕J),'''S'
      ''
      (⍕Z),('|'),N2NP I2NP Z
    L2:→(J=⌈/S)/0
      A←Z
      →(TONSYS≥J←J+1)/L1
    ∇
```

Fig. 8-13. Function SIGMAS builds all chords of particular sizes containing just the intervals in I.

Listing 8-3 are a pair of numbers. The first of these gives the number of unique pitch names in the total chord. The second measures the *tension* of the structure. All the chords have been sorted in order of increasing tension. The topic of tension is too important to describe incidentally under the heading of "Raw Material." It will be presented subsequently in its own right. Let me just add here that these tension assignments were based on particular assumptions about the instrumentation to be used for each stratum. Different assumptions could rearrange the order of the structures somewhat, but I think you'll find on playing them as written with piano or strings that the idea of "increasing tension" does indeed come across as you progress down the list.

SELECTION SCHEMES BASED ON ARRAYS

The sample techniques shown above were said to be based on intervals. In a broader sense, they were based on numeric vectors. Other methods, still based on vectors, could be shown for cataloging "scales" of tone color or "structures" of synthesizer control settings. There are elements of composition that are just too involved to be controlled by a single set of values, however, so instead we will turn to more complex techniques for generating raw material.

Schillinger prescribed a direct way to generate

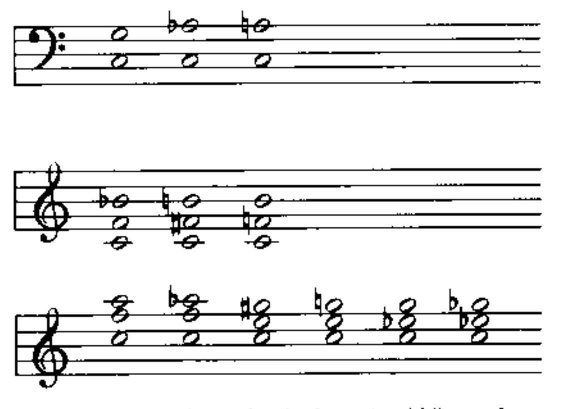


Fig. 8-14. Allowed forms for the lowest, middle, and upper strata (top, second, and bottom lines, respectively) depicted in Listing 8-3.

harmonic continuity using two sequences of numbers. The composer defines a set of chord structures and assigns an identifying number to each chord. This allows a single element of one vector to select a chord while the corresponding element of a second vector governs the choice of the root tone for the chord. As elements of both vectors are cycled simultaneously in this way, a harmonic sequence emerges whose length depends on the lengths of the two vectors and on the "interference" they generate between structure types and root tones. When first introduced to this procedure, composers are fascinated by the unexpected and unusual nature of the resulting harmonic continuity. But with experience comes the realization that the two arbitrarily chosen vectors cannot give the composer a satisfying degree of stylistic control.

A revision of Schillinger's method, using an array instead of two vectors, can restore the composer's control over his style. First, as Schillinger suggested (though in different terms), an array is established in which each row constains the intervals of one of the chosen harmonic structures. Notice, in choosing these structures, one aspect of the style becomes fixed. Say we set HS equal to the array

```
4  3
3  4
4  4
5  5
```

The entries here are intervals, so the first three rows define what are normally called *major, minor,* and *augmented triads*. The last row specifies a triad built of two perfect fourths. By adding a third column (interval), we could describe four-note chords.

Now, departing from Schillinger's procedure, the composer is to construct another array which will determine the other component of style. Just as the above array shows the composer's choice of chordal types, this new one will exhibit his preferences for the ways that chords may be connected in time. Suppose we set PREF to the array

```
0  1  ¯4
1  1  ¯4
1  0   2
```

The first two columns refer to row indices of the structure array (HS), and the last column describes a root-tone motion. The first row here implies that a structure of the type in row zero of HS may go to one of the type in row one if the root tone moves down four semitones (a major third). The second row allows the kind of structure in row one to be repeated if the root moves down four semitones. If the root moves up two semitones, the third row permits the type of chord in row one to change to the type in row zero.

Even with such small arrays, we need a program to maintain the proper synchronization of elements in the total procedure. At times the composer will want to give priority to certain root-tone movements. At other times his thoughts on the ordering of consecutive structures will predominate. There is even a chance that he may want to play Schillinger's game at other times by describing both sequences simultaneously and independently. In the latter case, the PREF array plays no role. But in either of the other situations, the program will begin at the top of the PREF array and work downward until it finds the first row that will satisfy the current root-tone motion or two-structure sequence. In compositional use, both arrays can be altered from time to time in various ways. (We'll demonstrate this later.) But for cataloguing, their forms should be fixed in order to collect the complete root/structure sequence implied.

With three ways to define sequences (by structure, by root, or by both simultaneously), you might expect to require three programs. But the existing limits on my talent for selecting mnemonics, not to mention the limits of my memory, made it necessary to introduce a fourth. All I need to remember now is the name **HCON**, suggesting Harmonic CONtinuity. The functions are listed in Fig. 8-15.

After typing **HCON** I ask myself, in my

```
      ∇ HCON;CYC;STR;IRC;IRS
 [1]  L0:IRC←IRS←⍳0
 [2]   'ENTER CYCLE VECTOR'
 [3]   CYC←,⎕
 [4]   →(0<ρCYC)↑L1
 [5]   'ENTER INITIAL ROOT AND DFLT CYCLE'
 [6]   IRC←2↑⎕
 [7]  L1:'ENTER STRCTR VCTR'
 [8]   STR←,⎕
 [9]   →(0<ρSTR)↑L2
 [10]  →(0<ρIRC)↑L5
 [11]  'ENTER INITIAL RT AND STR'
 [12]  IRS←2↑⎕
 [13] L2:'PRESS RETURN TO CNTU, NON-BLANK TO QUIT'
 [14]  →((0≠ρIRS),0≠ρIRC)/L3,L4
 [15]  CYC CSHCON STR
 [16]  →
 [17] L3:IRS CHCON CYC
 [18]  →
 [19] L4:IRC SHCON STR
 [20]  →
 [21] L5:'THERE MUST BE AT LEAST ONE SEQUENCE'
 [22]  →L0
      ∇

      ∇ IRS CHCON CYC;SO;SN;I;RT;TP;TC;Z
 [1]   I←0
 [2]   SO←IRS[1]
 [3]   RT←IRS[0]
 [4]   Z←(1,ρZ)ρZ←+\RT,HS[SO;]
 [5]   TP←TONSYS|PREF[;2]
 [6]  L1:TC←TONSYS|CYC[I]
 [7]   SN←1↑PREF[((PREF[;0]=SO)∧TP=TC)/⍳1↑ρPREF;1],IRS[1]
 [8]   RT←RT+CYC[I]
 [9]   Z←Z,[0]+\RT,HS[SN;]
 [10]  SO←SN
 [11]  →(0≠I←(ρ,CYC)|I+1)↑L1
 [12]  N2NP Z
 [13]  Z←(0,¯1↑ρZ)ρZ
 [14]  →(0=ρ,⍞)↑L1
      ∇

     ∇ IRC SHCON STR;SO;SN;I;RT;Z
[1]   I←1
[2]   SO←STR[0]
[3]   RT←IRC[0]
[4]   Z←(1,ρZ)ρZ←+\RT,HS[SO;]
[5]  L1:SN←STR[I]
[6]   CYC←1↑PREF[(∧/PREF[;⍳2]=((1↑ρPREF),2)ρSO,SN)/⍳1↑ρPREF;2],IRC[1]
[7]   RT←RT+CYC
```

```
[8]    Z←Z,[0]+\RT,HS[SN;]
[9]    SO←SN
[10]   →(0≠I←(ρ,STR)|I+1)↑L1
[11]   N2NP Z
[12]   Z←(0,¯1↑ρZ)ρZ
[13]   →(0=ρ,⍞)↑L1
     ∇



     ∇ CYC CSHCON STR;K;SO;Z
[1]    SO←0
[2]    K←×/(ρ,STR),ρ,CYC
[3]    Z←I2NP HS[KρSTR;]
[4]   L1:N2NP Z+⍉(⌽ρZ)ρ1↓+\SO,KρCYC
[5]    SO←+/SO,KρCYC
[6]    →(0=ρ,⍞)↑L1
     ∇



     ∇ Z←I2NP A
[1]    Z←0,TONSYS|+\A
     ∇
```

Fig. 8-15. A set of functions for creating sequences of harmonic structures.

typically laconic way, to "**ENTER CYCLE VECTOR.**" When the programmer and user are the same person, such behavior goes unnoticed, but now I must explain that message. The program should be asking the user to enter a numeric vector whose elements will specify the intervals between successive root tones. If you were to type

¯3 ¯4 2 ¯4 5,

assuming the initial root to be C, you would be constructing the root tone sequence

C A F G E♭ A♭ F D♭ E♭ B E C♯ A B G C.

After cycling over the vector three times, we arrive back at the first root, C. If, instead of complying with the program's request, you entered an empty vector (ι0), the program would assume you want to describe only a sequence of structures leaving the algorithm to select the root-tone intervals. But to do so, it must have a starting point, and so it asks for an "**INITIAL ROOT AND DFLT CYCLE.**" For an initial root tone, you must enter an integer representing the pitch (zero for C, one for C♯, etc.). This must be followed (on the same line) by another integer which will be used by default for the root-tone motion whenever the PREF array fails to account for consecutive structures.

In either case—whether you have selected a sequence of root-tone intervals or just an initial root tone and a default interval—the program will then ask you to "**ENTER STRCTR VCTR,**" meaning that it expects a sequence of row indices referring back to the harmonic-structure array HS. It will also accept an empty vector here in case you want only your root-tone intervals to be in control. Now, it must check which logical path is to be taken. If a sequence of structures has been entered, it branches to L2, where you are asked to confirm that you want to continue. If you press the Return key, meaning that you do wish to go in, it must then

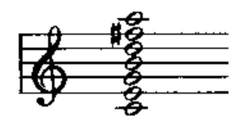
decide whether to call **CSHCON** to compute the results based on the simultaneous use of both vectors, or to call **SHCON** so that the initial root and cycle, the structure vector, and the PREF array will be used to compute a harmonic continuity.

On the other hand, should the structure vector you have just entered be found empty—meaning you typed ⍳0—the program must now test to see whether a sequence of root-tone intervals exists. If it, too, is empty, you will be informed "**THERE MUST BE AT LEAST ONE SEQUENCE**" and then find yourself back at the beginning being asked to "**ENTER CYCLE VECTOR.**" But if root-tone intervals have been specified, you will be asked to enter an initial root tone and structure ("**ENTER INITIAL RT AND STR**"). This structure will serve for the first chord as well as a default structure whenever PREF cannot satisfy an interval specification.

Each chord in the calculated continuity will then be spelled out on a separate line. When one cycle of the input vector(s) has been used, the program will wait for a signal to continue or stop. If an empty line is entered, another cycle is calculated using the end of the previous one for reference. Entering any character will end execution.

The arrays, HS and PREF, given above were used to generate the page of output shown in Fig. 8-16. (Compare this with the harmony of the example shown in Fig. 7-21!)

I mentioned the possibility of having additional columns in the harmonic-structure array HS. In fact, HS can hold structures containing more pitches than desired for single chords. (We'll see why when we take up tonality.) A row of HS with the entries 4 3 4 3 4 3 would represent this seven-note structure:



Now, a pitch-selector array, PS, can be selected to limit chords to the desired number of voices. For example, we might construct PS so that eight different four-note chords could be extracted from a seven-note structure:

```
       0 0 1 2
       1 0 1 2
       2 0 1 2
PS =   0 1 2 3
       0 1 3 4
       0 1 3 5
       0 3 4 5
       0 2 4 6
```

A root tone R and a row of HS, say the Ith row, would define a seven-note structure through the operation:

```
+ \ R,HS[I;]
```

The Jth row of PS would narrow this down to the chord:

```
(+ \ R,HS[I;])[PS[J;]]
```

Using each row of the pitch selector successively on the structure shown above would create the chords shown in Fig. 8-17.

To allow simultaneous selection of harmonic structure, chordal tones, and root progression, a third vector could now be incorporated into the procedure. You will need new **HCON** functions and preference arrays to account for the various relationships you want between the vectors. That is, you might define a single vector and then use a PREF array to select both of the remaining elements. Or one PREF array could select one of the elements and a different array the other. Notice, you could relate structures and cyles in one array (as above), structures and chordal tones in another, and even cycles and chordal tones in a third. Calling the three vectors A, B, and C for simplicity, you could define A and then choose either the array that relates A to B or the one connecting A to C. If the first were selected, you would then need to decide whether to use the PREF array that derives C from A or the one that gets C from B. In addition, you could define any two of the three vectors and de-

```
        HCON
ENTER CYCLE VECTOR
□:
        10
ENTER INITIAL ROOT AND DEFAULT CYCLE
□:
        0 5
ENTER STRUCTURE VECTOR
□:
0 1 1 0 1 1 1
WHEN KYBD OPENS, PRESS RETURN TO CONTINUE, NON-BLANK TO QUIT

 C      E      G
 G↑     B      D↑
 E      G      B
 F↑     A↑     C↑
 D      F      A
 A↑     C↑     F
 F↑     A      C↑

 G↑     C      D↑
 E      G      B
 C      D↑     G
 D      F↑     A
 A↑     C↑     F
 F↑     A      C↑
 D      F      A

 E      G↑     B
 C      D↑     G
 G↑     B      D↑
 A↑     D      F
 F↑     A      C↑
 D      F      A
 A↑     C↑     F
X
```

Fig. 8-16. An example using function HCON.

rive the third from either or both of the given ones. You could also select Schillinger's approach by defining all three vectors, thereby letting the computer carry out the accounting without the control over style afforded by PREF arrays.

Instead of or in addition to the pitch-selection vector, you could use a voicing selector. VS might have the form;

1 2 0
2 0 1
2 1 0

implying the use of three-note chords. The entries in each row determine the way the voices in one chord are to move to those in the next. The column index of an entry applies to a tone in the first chord, while the entry itself determines the tone to which the first one must move in the next. Thus a one in column zero says that the voice playing the zeroth entry in the last harmonic structure will next play the note whose index is one in the new harmonic structure. The first line of Fig. 8-18 shows the first seven chords from Fig. 8-16 transcribed directly (without voicing) to notation. The second line con-

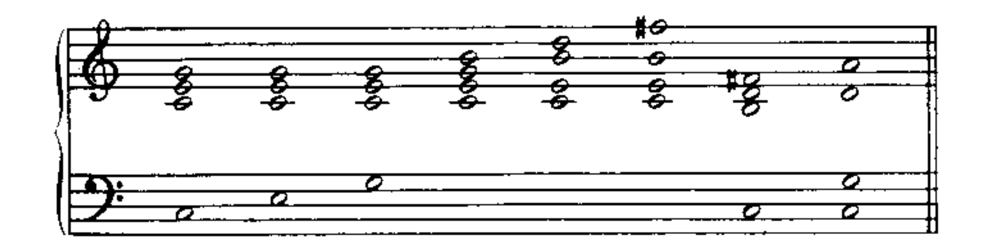


Fig. 8-17. A seven-note structure containing the intervals 4 3 4 3 4 3.

sists of the same chords after applying the rows of the voicing selector array shown above in sequence, i.e., row zero is used to go from the first chord to the second, row one for the transition from the second to the third, row two for the third to the fourth, back to row zero for the next chordal change, and so on.

Notice that PREF arrays can now be used to relate root-tone interval, chord structure, and/or pitch selection to voicing selection! Each gives a different kind of control over style. But most important, catalogs of such material appear to be "selected" rather than "generated" because of their stylized contents!

There are other possibilities that can be treated in much the same way. The one that immediately comes to mind plays a major role in style—patterns of accompaniment. But this would take us further into the subject of orchestration and far ahead of the topics of composition yet to be discussed. Before you begin pondering the other possibilities, it would be wise to think about other methods of your own design for collecting the kind of material described above.

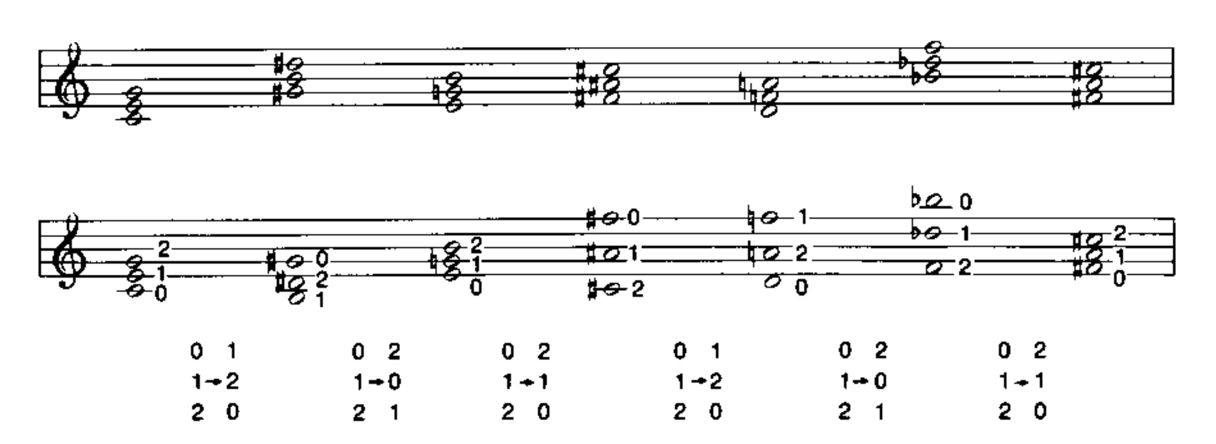


Fig. 8-18. The top line shows the first seven chords produced by function HCON in Fig. 8-16. The second line illustrates applying consecutive rows of the voicing selector array to the chords on the top line.

Listing 8-1. Using function SIGMAS to find all four-note and nine-note chords containing only major and minor thirds.

```
       SIGMAS 3 4
 ENTER VECTOR OF SIGMAS TO BE LISTED
 ⎕:



       4 9
       INTERVALS:  3 4
 ***********************************************************
 7 SIGMA 4'S
  3 3 3| C    D↑   F↑   A
  3 3 4| C    D↑   F↑   A↑
  3 4 3| C    D↑   G    A↑
  3 4 4| C    D↑   G    B
  4 3 3| C    E    G    A↑
  4 3 4| C    E    G    B
  4 4 3| C    E    G↑   B
 ***********************************************************


 35 SIGMA 9'S
  3 3 3 4 3 3 3 4| C    D↑   F↑   A    C↑   E    G    A↑   D
  3 3 3 4 3 3 4 3| C    D↑   F↑   A    C↑   E    G    B    D
  3 3 3 4 3 4 3 3| C    D↑   F↑   A    C↑   E    G↑   B    D
  3 3 3 4 4 3 3 3| C    D↑   F↑   A    C↑   F    G↑   B    D
  3 3 4 3 3 3 4 3| C    D↑   F↑   A↑   C↑   E    G    B    D
  3 3 4 3 3 4 3 3| C    D↑   F↑   A↑   C↑   E    G↑   B    D
  3 3 4 3 4 3 3 3| C    D↑   F↑   A↑   C↑   F    G↑   B    D
  3 3 4 4 3 4 4 3| C    D↑   F↑   A↑   D    F    A    C↑   E
  3 4 3 3 3 4 3 3| C    D↑   G    A↑   C↑   E    G↑   B    D
  3 4 3 3 4 3 3 3| C    D↑   G    A↑   C↑   F    G↑   B    D
  3 4 3 4 3 4 4 3| C    D↑   G    A↑   D    F    A    C↑   E
  3 4 3 4 4 3 4 3| C    D↑   G    A↑   D    F↑   A    C↑   E
  3 4 3 4 4 3 4 4| C    D↑   G    A↑   D    F↑   A    C↑   F
  3 4 4 3 3 4 4 3| C    D↑   G    B    D    F    A    C↑   E
  3 4 4 3 4 3 4 3| C    D↑   G    B    D    F↑   A    C↑   E
  3 4 4 3 4 3 4 4| C    D↑   G    B    D    F↑   A    C↑   F
  3 4 4 3 4 4 3 3| C    D↑   G    B    D    F↑   A↑   C↑   E
  3 4 4 3 4 4 3 4| C    D↑   G    B    D    F↑   A↑   C↑   F
  4 3 3 3 4 3 3 3| C    E    G    A↑   C↑   F    G↑   B    D
  4 3 3 3 4 3 3 4| C    E    G    A↑   C↑   F    G↑   B    D↑
  4 3 3 4 3 3 3 4| C    E    G    A↑   D    F    G↑   B    D↑
  4 3 3 4 4 3 4 4| C    E    G    A↑   D    F↑   A    C↑   F
  4 3 4 3 4 3 4 4| C    E    G    B    D    F↑   A    C↑   F
  4 3 4 3 4 4 3 4| C    E    G    B    D    F↑   A↑   C↑   F
  4 3 4 4 3 3 4 4| C    E    G    B    D↑   F↑   A    C↑   F
  4 3 4 4 3 4 3 4| C    E    G    B    D↑   F↑   A↑   C↑   F
  4 3 4 4 3 4 4 3| C    E    G    B    D↑   F↑   A↑   D    F
  4 4 3 3 4 3 4 4| C    E    G↑   B    D    F↑   A    C↑   F
  4 4 3 3 4 4 3 4| C    E    G↑   B    D    F↑   A↑   C↑   F
```

```
4 4 3 4 3 3 4 4| C    E    G↑   B    D↑   F↑   A    C↑   F
4 4 3 4 3 4 3 4| C    E    G↑   B    D↑   F↑   A↑   C↑   F
4 4 3 4 3 4 4 3| C    E    G↑   B    D↑   F↑   A↑   D    F
4 4 3 4 4 3 3 4| C    E    G↑   B    D↑   G    A↑   C↑   F
4 4 3 4 4 3 4 3| C    E    G↑   B    D↑   G    A↑   D    F
4 4 3 4 4 3 4 4| C    E    G↑   B    D↑   G    A↑   D    F↑
```

Listing 8-2. Finding all 12-note chords built in thirds (3,4), and in minor thirds, perfect fourths, and augmented fourths (3,5,6).

```
      SIGMAS 3 4
ENTER VECTOR OF SIGMAS TO BE LISTED
⎕:
      12
      INTERVALS:  3 4
************************************************************
4 SIGMA 12'S

 3 3 3 4 3 3 3 4 3 3 3|
 C    D↑   F↑   A    C↑   E    G    A↑   D    F    G↑   B
 4 3 3 4 3 3 3 4 3 3 4|
 C    E    G    A↑   D    F    G↑   B    D↑   F↑   A    C↑
 4 3 3 4 4 3 4 4 3 3 4|
 C    E    G    A↑   D    F↑   A    C↑   F    G↑   B    D↑
 4 4 3 4 4 3 4 4 3 4 4|
 C    E    G↑   B    D↑   G    A↑   D    F↑   A    C↑   F




      SIGMAS 3 5 6
ENTER VECTOR OF SIGMAS TO BE LISTED
⎕:
      12
      INTERVALS:  3 5 6
************************************************************
92 SIGMA 12'S
 3 3 3 5 3 3 3 5 3 3 3|
 C    D↑   F↑   A    D    F    G↑   B    E    G    A↑   C↑
 3 3 3 5 3 3 3 5 6 3 6|
 C    D↑   F↑   A    D    F    G↑   B    E    A↑   C↑   G
 3 3 3 5 6 3 6 5 3 3 3|
 C    D↑   F↑   A    D    G↑   B    F    A↑   C↑   E    G
 3 3 3 5 6 3 6 5 6 3 6|
 C    D↑   F↑   A    D    G↑   B    F    A↑   E    G    C↑
 3 3 5 3 3 3 5 6 3 6 5|
 C    D↑   F↑   B    D    F    G↑   C↑   G    A↑   E    A
 3 3 5 5 5 5 3 3 5 6 3|
 C    D↑   F↑   B    E    A    D    F    G↑   C↑   G    A↑
 3 3 5 6 3 5 3 5 5 5 3|
 C    D↑   F↑   B    F    G↑   C↑   E    A    D    G    A↑
 3 3 5 6 3 5 6 3 6 5 5|
```

C	D↑	F↑	B	F	G↑	C↑	G	A↑	E	A	D
3 3	5	6 3	6 5	3 3	3 5|						
C	D↑	F↑	B	F	G↑	D	G	A↑	C↑	E	A
3 3	5	6 5	6 5	5 6	5 6|						
C	D↑	F↑	B	F	A↑	E	A	D	G↑	C↑	G
3 5	3	3 3	5 6	3 6	5 3|						
C	D↑	G↑	B	D	F	A↑	E	G	C↑	F↑	A
3 5	3	5 3	3 3	5 3	5 3|						
C	D↑	G↑	B	E	G	A↑	C↑	F↑	A	D	F
3 5	3	5 3	6 5	3 5	3 5|						
C	D↑	G↑	B	E	G	C↑	F↑	A	D	F	A↑
3 5	3	6 5	6 3	6 5	3 5|						
C	D↑	G↑	B	F	A↑	E	G	C↑	F↑	A	D
3 5	3	6 5	6 5	5 5	6 5|						
C	D↑	G↑	B	F	A↑	E	A	D	G	C↑	F↑
3 5	5	5 3	5 3	6 5	3 3|						
C	D↑	G↑	C↑	F↑	A	D	F	B	E	G	A↑
3 5	5	5 5	6 5	6 5	5 5|						
C	D↑	G↑	C↑	F↑	B	F	A↑	E	A	D	G
3 5	6	3 6	5 3	3 3	5 3|						
C	D↑	G↑	D	F	B	E	G	A↑	C↑	F↑	A
3 5	6	5 6	5 5	6 5	6 5|						
C	D↑	G↑	D	G	C↑	F↑	B	F	A↑	E	A
3 6	5	3 3	3 5	3 3	3 5|						
C	D↑	A	D	F	G↑	B	E	G	A↑	C↑	F↑
3 6	5	3 3	5 5	5 5	3 3|						
C	D↑	A	D	F	G↑	C↑	F↑	B	E	G	A↑
3 6	5	6 3	6 5	6 3	6 5|						
C	D↑	A	D	G↑	B	F	A↑	E	G	C↑	F↑
3 6	5	6 5	5 5	6 5	6 3|						
C	D↑	A	D	G↑	C↑	F↑	B	F	A↑	E	G
5 3	3	3 5	3 3	3 5	6 3|						
C	F	G↑	B	D	G	A↑	C↑	E	A	D↑	F↑
5 3	3	3 5	6 3	6 5	3 3|						
C	F	G↑	B	D	G	C↑	E	A↑	D↑	F↑	A
5 3	3	5 6	5 3	3 5	5 6|						
C	F	G↑	B	E	A↑	D↑	F↑	A	D	G	C↑
5 3	3	5 6	5 6	5 5	6 5|						
C	F	G↑	B	E	A↑	D↑	A	D	G	C↑	F↑
5 3	5	3 5	5 5	3 5	3 5|						
C	F	G↑	C↑	E	A	D	G	A↑	D↑	F↑	B
5 3	5	3 5	6 3	5 3	5 3|						
C	F	G↑	C↑	E	A	D↑	F↑	B	D	G	A↑
5 3	5	5 5	3 5	3 6	5 6|						
C	F	G↑	C↑	F↑	B	D	G	A↑	E	A	D↑
5 3	5	5 5	5 3	3 5	6 5|						
C	F	G↑	C↑	F↑	B	E	G	A↑	D↑	A	D
5 3	5	5 5	5 5	5 5	3 5|						
C	F	G↑	C↑	F↑	B	E	A	D	G	A↑	D↑
5 3	5	5 5	5 6	5 6	5 5|						
C	F	G↑	C↑	F↑	B	E	A↑	D↑	A	D	G
5 3	5	6 3	5 3	5 5	5 5|						
C	F	G↑	C↑	G	A↑	D↑	F↑	B	E	A	D
5 3	5	6 3	6 5	6 3	5 3|						

```
C     F     G↑    C↑    G     A↑    E     A     D↑    F↑    B     D
5 3 6 5 3 3 3 5 6 3 5|
C     F     G↑    D     G     A↑    C↑    E     A     D↑    F↑    B
5 3 6 5 3 3 5 5 5 5 6|
C     F     G↑    D     G     A↑    C↑    F↑    B     E     A     D↑
5 3 6 5 6 5 5 5 6 5 6|
C     F     G↑    D     G     C↑    F↑    B     E     A↑    D↑    A
5 5 3 5 5 5 5 6 5 6 5|
C     F     A↑    C↑    F↑    B     E     A     D↑    G↑    D     G
5 5 5 3 3 5 6 3 5 3 6|
C     F     A↑    D↑    F↑    A     D     G↑    B     E     G     C↑
5 5 5 3 5 5 5 5 6 5 6|
C     F     A↑    D↑    F↑    B     E     A     D     G↑    C↑    G
5 5 5 5 3 5 3 6 5 3 5|
C     F     A↑    D↑    G↑    B     E     G     C↑    F↑    A     D
5 5 5 5 3 5 5 5 5 6 5|
C     F     A↑    D↑    G↑    B     E     A     D     G     C↑    F↑
5 5 5 5 5 5 5 5 5 5 5|
C     F     A↑    D↑    G↑    C↑    F↑    B     E     A     D     G
5 5 5 5 6 5 6 5 5 5 5|
C     F     A↑    D↑    G↑    D     G     C↑    F↑    B     E     A
5 5 5 6 5 6 3 5 3 6 5|
C     F     A↑    D↑    A     D     G↑    B     E     G     C↑    F↑
5 5 5 6 5 6 5 5 5 5 3|
C     F     A↑    D↑    A     D     G↑    C↑    F↑    B     E     G
5 5 6 3 6 5 3 6 5 3 3|
C     F     A↑    E     G     C↑    F↑    A     D↑    G↑    B     D
5 5 6 5 6 3 5 3 6 5 6|
C     F     A↑    E     A     D↑    F↑    B     D     G↑    C↑    G
5 5 6 5 6 5 3 3 5 6 5|
C     F     A↑    E     A     D↑    G↑    B     D     G     C↑    F↑
5 5 6 5 6 5 5 5 5 3 5|
C     F     A↑    E     A     D↑    G↑    C↑    F↑    B     D     G
5 5 6 5 6 5 6 5 6 5 5|
C     F     A↑    E     A     D↑    G↑    D     G     C↑    F↑    B
5 6 3 5 3 6 5 6 5 5 5|
C     F     B     D     G     A↑    E     A     D↑    G↑    C↑    F↑
5 6 3 6 5 3 3 3 5 3 3|
C     F     B     D     G↑    C↑    E     G     A↑    D↑    F↑    A
5 6 3 6 5 6 3 6 5 6 3|
C     F     B     D     G↑    C↑    G     A↑    E     A     D↑    F↑
5 6 5 3 3 3 5 3 6 5 6|
C     F     B     E     G     A↑    C↑    F↑    A     D↑    G↑    D
5 6 5 3 3 5 3 3 5 6 5|
C     F     B     E     G     A↑    D↑    F↑    A     D     G↑    C↑
5 6 5 3 3 5 5 5 5 3 5|
C     F     B     E     G     A↑    D↑    G↑    C↑    F↑    A     D
5 6 5 3 3 5 6 5 6 5 5|
C     F     B     E     G     A↑    D↑    A     D     G↑    C↑    F↑
5 6 5 5 5—5 3 5 5 5 5|
C     F     B     E     A     D     G     A↑    D↑    G↑    C↑    F↑
5 6 5 5 5 6 5 6 3 5 3|
C     F     B     E     A     D     G↑    C↑    G     A↑    D↑    F↑
5 6 5 5 6 5 6 5 3 3 5|
```

```
C     F     B     E     A     D↑    G↑    D     G     A↑    C↑    F↑
5  6  5  6  3  5  3  6  5  6  5|
C     F     B     E     A↑    C↑    F↑    A     D↑    G↑    D     G
5  6  5  6  5  3  3  5  6  5  6|
C     F     B     E     A↑    D↑    F↑    A     D     G↑    C↑    G
5  6  5  6  5  5  5  5  3  5  5|
C     F     B     E     A↑    D↑    G↑    C↑    F↑    A     D     G
5  6  5  6  5  5  6  5  6  5  3|
C     F     B     E     A↑    D↑    G↑    D     G     C↑    F↑    A
6  3  5  3  5  5  5  3  5  3  6|
C     F↑    A     D     F     A↑    D↑    G↑    B     E     G     C↑
6  3  5  3  6  5  3  3  5  5  5|
C     F↑    A     D     F     B     E     G     A↑    D↑    G↑    C↑
6  3  5  3  6  5  6  5  5  5  6|
C     F↑    A     D     F     B     E     A↑    D↑    G↑    C↑    G
6  3  5  5  6  3  6  5  5  3  6|
C     F↑    A     D     G     C↑    E     A↑    D↑    G↑    B     F
6  3  6  5  3  3  3  5  3  3  3|
C     F↑    A     D↑    G↑    B     D     F     A↑    C↑    E     G
6  3  6  5  3  3  3  5  6  3  6|
C     F↑    A     D↑    G↑    B     D     F     A↑    E     G     C↑
6  3  6  5  6  3  6  5  3  3  3|
C     F↑    A     D↑    G↑    D     F     B     E     G     A↑    C↑
6  3  6  5  6  3  6  5  6  3  6|
C     F↑    A     D↑    G↑    D     F     B     E     A↑    C↑    G
6  5  3  3  3  5  3  3  3  5  6|
C     F↑    B     D     F     G↑    C↑    E     G     A↑    D↑    A
6  5  3  3  3  5  6  3  6  5  6|
C     F↑    B     D     F     G↑    C↑    G     A↑    E     A     D↑
6  5  3  3  5  6  5  6  5  5  6|
C     F↑    B     D     F     A↑    E     A     D↑    G↑    C↑    G
6  5  5  3  3  5  6  5  3  3  5|
C     F↑    B     E     G     A↑    D↑    A     D     F     G↑    C↑
6  5  5  5  5  3  3  5  6  3  5|
C     F↑    B     E     A     D     F     G↑    C↑    G     A↑    D↑
6  5  5  5  5  3  5  5  5  5  6|
C     F↑    B     E     A     D     F     A↑    D↑    G↑    C↑    G
6  5  5  5  6  5  6  3  5  3  6|
C     F↑    B     E     A     D↑    G↑    D     F     A↑    C↑    G
6  5  5  6  5  6  5  3  3  5  6|
C     F↑    B     E     A↑    D↑    A     D     F     G↑    C↑    G
6  5  6  3  5  3  3  3  5  6  5|
C     F↑    B     F     G↑    C↑    E     G     A↑    D↑    A     D
6  5  6  3  5  3  5  5  5  3  5|
C     F↑    B     F     G↑    C↑    E     A     D     G     A↑    D↑
6  5  6  3  5  3  6  5  6  5  5|
C     F↑    B     F     G↑    C↑    E     A↑    D↑    A     D     G
6  5  6  3  6  5  3  3  3  5  6|
C     F↑    B     F     G↑    D     G     A↑    C↑    E     A     D↑
6  5  6  3  6  5  6  3  6  5  6|
C     F↑    B     F     G↑    D     G     C↑    E     A↑    D↑    A
6  5  6  5  3  3  5  6  5  6  5|
C     F↑    B     F     A↑    C↑    E     A     D↑    G↑    D     G
6  5  6  5  5  5  5  3  5  5  5|
```

```
C     F↑    B     F     A↑    D↑    G↑    C↑    E     A     D     G
6 5 6 5 5 5 6 5 6 3 5|
C     F↑    B     F     A↑    D↑    G↑    D     G     C↑    E     A
6 5 6 5 5 6 5 6 5 3 3|
C     F↑    B     F     A↑    D↑    A     D     G↑    C↑    E     G
6 5 6 5 6 5 6 5 6 5 6|
C     F↑    B     F     A↑    E     A     D↑    G↑    D     G     C↑
```

Listing 8-3. Catalog of all 324 structures formed from the strata shown in Fig. 8-14, sorted in order of increasing tension.

C	A	E	A	D	F↑	A	C	:	5	22
C	G	D	G	C	E	G	A↑	:	5	23
C	A	E	A	D	G	C	E	:	5	24
C	G	D	G	C	F	A	C	:	5	34
C	G	D	G	C	F	A↑	D	:	5	35
C	G	D	G	C	E	A	C	:	5	42
C	A	D↑	A	D	F↑	A	C	:	5	111
C	G↑	D	G	C	F	A↑	D	:	6	144
C	A	E	A	D	G	B	D	:	6	144
C	A	E	A	D	F↑	B	D	:	6	152
C	A	E	A	D↑	G	C	D↑	:	5	202
C	G	D	G	C	E	G	B	:	5	221
C	G↑	E	A↑	D↑	G↑	C	D↑	:	5	222
C	G	D↑	A	D	F↑	A	C	:	6	231
C	G↑	D	G	C	E	G	A↑	:	6	233
C	G↑	D↑	G↑	D	F↑	A↑	D	:	6	243
C	G↑	E	A↑	D↑	G	C	D↑	:	6	321
C	G↑	D↑	G↑	D	F↑	B	D	:	6	322
C	A	E	A	D↑	G	B	D↑	:	6	322
C	G↑	E	A↑	D↑	G	A↑	D	:	7	333
C	A	E	A	D↑	G	B	D	:	7	333
C	A	D↑	A	D	F↑	B	D	:	6	340

C	G	D↑	A	D	G	B	D	:	6	341
C	A	D↑	A	D	G	B	D	:	6	342
C	A	E	A	D↑	G↑	C	D↑	:	5	400
C	G↑	D	G	C	E	G	B	:	6	421
C	A	F	B	E	G↑	B	D	:	7	431
C	A	D↑	G↑	D	F↑	B	D	:	7	431
C	G	D↑	A	D	F↑	B	D	:	7	440
C	G↑	D↑	G↑	D	G	A↑	D	:	6	441
C	G↑	D↑	G↑	D	G	B	D	:	6	520
C	A	E	A	D↑	G↑	B	D↑	:	6	520
C	A	F	B	E	G↑	B	D↑	:	7	521
C	A	D↑	G↑	D	G	B	D	:	7	531
C	A	E	A	D↑	G↑	B	D	:	7	531
C	A	F	A↑	D↑	G	C	D↑	:	6	1024
C	G	C↑	G	C	E	G	A↑	:	5	1102
C	A	E	A	D↑	G	C	E	:	5	1102
C	G↑	D↑	G↑	C↑	F	G↑	C	:	5	1111
C	G↑	D↑	G↑	C↑	F	G↑	B	:	6	1112
C	G↑	D	G	C	F	G↑	C	:	5	1113
C	A	D↑	A	D	G	C	D↑	:	5	1113
C	G↑	D↑	G↑	C↑	F↑	B	D↑	:	6	1114
C	A	E	A	D	G	C	D↑	:	6	1114
C	A	F	A↑	D↑	G↑	C	D↑	:	6	1114
C	A	E	A	D	F↑	A	C↑	:	6	1121
C	G	D↑	A	D	G	C	D↑	:	5	1122
C	G↑	D	G	C	E	G↑	C	:	5	1122
C	G↑	E	A↑	D↑	G↑	C	E	:	5	1122
C	G↑	E	A	D	G	C	E	:	6	1123
C	G↑	D	G	C	F	A	C	:	6	1123
C	A	D↑	A	D	G	C	E	:	6	1123
C	G	D↑	A	D	G	C	E	:	6	1132
C	G↑	D	G	C	E	A	C	:	6	1132
C	G↑	E	A	D	F↑	B	D	:	7	1143

C	G↑	D	G↑	C↑	F	G↑	C	:	5	1211
C	G↑	D	G↑	C↑	F	G↑	B	:	6	1211
C	A	D↑	A	D	F↑	A	C↑	:	6	1211
C	G↑	D↑	G↑	D	F↑	B	D↑	:	6	1212
C	A	E	A↑	D↑	G	C	D↑	:	6	1212
C	A	D↑	G↑	D	F↑	A	C	:	6	1212
C	G	C↑	G	C	E	A	C	:	5	1220
C	G↑	E	A↑	D↑	G	C	E	:	6	1221
C	G	C↑	G	C	F	A	C	:	5	1222
C	A	F	B	E	A	D	F	:	6	1222
C	G↑	E	A↑	D↑	G	A↑	C↑	:	7	1223
C	A	D↑	G↑	C↑	F↑	B	D↑	:	7	1223
C	A	D↑	A	D	F↑	B	D↑	:	6	1230
C	G	D↑	A	D	G	B	D↑	:	6	1231
C	G	D↑	A	D	G	A↑	D	:	6	1232
C	A	F	B	E	A	D	F↑	:	7	1232
C	A	D↑	A	D	G	B	D↑	:	6	1232
C	G	D	G	C↑	F	A	C	:	6	1233
C	G↑	E	A	D	G	B	D	:	7	1233
C	A	E	A	D	G	B	D↑	:	7	1233
C	G↑	D↑	G↑	D	F↑	A↑	C↑	:	7	1234
C	A	E	A	D	F↑	B	D↑	:	7	1241
C	G↑	E	A	D↑	G↑	C	D↑	:	5	1300
C	A	E	A	D↑	G↑	C	E	:	5	1300
C	G	C↑	G	C	E	G	B	:	5	1301
C	G↑	E	A	D↑	G	C	D↑	:	6	1301
C	G	C↑	F↑	C	E	A	C	:	6	1311
C	G↑	D↑	G↑	D	G	C	D↑	:	5	1311
C	A	D↑	G↑	C↑	F	G↑	C	:	6	1311
C	A	F	B	E	A	C	E	:	5	1311
C	A	F	A↑	E	A	C	E	:	5	1311
C	G↑	D	G	C	F	G↑	B	:	6	1312
C	G↑	E	A↑	D↑	G↑	B	D↑	:	6	1312
C	A	E	A↑	D↑	G↑	C	D↑	:	6	1312
C	A	F	B	E	A	C	D↑	:	6	1312
C	A	D↑	G↑	D	G	C	D↑	:	6	1312
C	A	F	A↑	E	G↑	C	E	:	6	1312
C	A	F	A↑	E	A	C	D↑	:	6	1312

C	A	F	A↑	E	G↑	C	D↑	:	7 1313
C	G↑	D	G	C	E	G↑	B	:	6 1321
C	G↑	D↑	G↑	D	G	C	E	:	6 1321
C	A	F	B	E	A	C↑	E	:	6 1321
C	A	D↑	G↑	D	F↑	B	D↑	:	7 1321
C	G	C↑	F↑	B	D↑	F↑	A	:	7 1322
C	A	D↑	G↑	C↑	F	G↑	B	:	7 1322
C	A	D↑	G↑	D	G	C	E	:	7 1322
C	G↑	E	A↑	D↑	G↑	B	D	:	7 1323
C	G	D↑	A	D	F↑	B	D↑	:	7 1330
C	G	D↑	A	D	F↑	A↑	D	:	7 1331
C	G	D↑	A	D	F↑	A	C↑	:	7 1331
C	G	D↑	G↑	D	G	A↑	D	:	6 1331
C	G↑	D↑	A	D	F↑	B	D	:	7 1331
C	G	D↑	G↑	D	F↑	A↑	D	:	7 1332
C	A	D↑	G↑	D	F↑	A↑	D	:	7 1332
C	G↑	D↑	G↑	D	G	B	D↑	:	6 1410
C	G↑	E	A	D↑	G↑	B	D↑	:	6 1410
C	A	F	B	E	G↑	C	E	:	6 1410
C	G	C↑	F↑	C	F	A	C	:	6 1411
C	G↑	E	A↑	D↑	G	B	D↑	:	7 1411
C	G↑	E	A	D↑	G	B	D↑	:	7 1411
C	A	F	B	E	G↑	C	D↑	:	7 1411
C	G	C↑	F↑	B	D↑	F↑	A↑	:	7 1412
C	G	D↑	G↑	D	G	B	D	:	6 1420
C	A	F	B	E	G↑	C↑	E	:	7 1420
C	G	D↑	G↑	D	F↑	B	D	:	7 1421
C	G↑	E	A	D↑	G↑	B	D	:	7 1421
C	A	D↑	G↑	D	G	B	D↑	:	7 1421
C	G↑	E	A↑	D↑	G	B	D	:	8 1422
C	G↑	E	A	D↑	G	B	D	:	8 1422
C	G	D	G	C↑	F↑	A	C	:	6 1431
C	G↑	D↑	A	D	G	B	D	:	7 1431
C	G↑	D↑	G↑	D	G	A↑	C↑	:	7 1432
C	A	D↑	G↑	D	G	A↑	D	:	7 1432
C	G	D	G	C	F	G↑	C	:	5 2014
C	G	D	G	C	E	G↑	C	:	5 2022

C	G↑	E	A	D	F↑	A	C	:	6 2023
C	G↑	D↑	G↑	C↑	F↑	A↑	C↑	:	6 2025
C	A	E	A	D	G	A↑	D	:	6 2025
C	G↑	D↑	G↑	C↑	F	A↑	C↑	:	6 2033
C	A	E	A	D	F↑	A↑	D	:	6 2033
C	G↑	E	A	D	F↑	A↑	D	:	7 2044
C	G↑	D↑	A	D	F↑	A	C	:	6 2112
C	A	E	A↑	D↑	G	C	E	:	6 2112
C	G↑	E	A↑	D↑	G↑	C↑	E	:	6 2113
C	G↑	D↑	G↑	D	F↑	A	C	:	6 2113
C	A	F	A↑	D↑	G↑	C↑	F	:	7 2114
C	G	D	G	C	F	A↑	C↑	:	6 2115
C	G	D	G	C↑	F	A↑	D	:	6 2123
C	G↑	E	A↑	D↑	G↑	C↑	F	:	7 2123
C	A	F	A↑	D↑	G	C	E	:	7 2123
C	G	C↑	G	C	F	A↑	D	:	6 2124
C	A	F	A↑	D↑	G	B	D↑	:	7 2124
C	A	F	A↑	D↑	G	A↑	D	:	7 2124
C	G	D	G	C	F	A	C↑	:	6 2133
C	G↑	D↑	G↑	C↑	F↑	A↑	D	:	7 2134
C	G↑	E	A	D	G	A↑	D	:	7 2134
C	G	D	G	C	E	A	C↑	:	6 2141
C	G↑	D	G↑	C↑	F	A↑	D	:	6 2141
C	G↑	D↑	G↑	C↑	F	A↑	D	:	7 2142
C	G↑	D	G↑	C↑	F↑	A↑	D	:	6 2143
C	G↑	E	A	D↑	G↑	C	E	:	5 2200
C	G↑	E	A	D↑	G	C	E	:	6 2201
C	A	D↑	G↑	C↑	F↑	A	C	:	6 2202
C	G	C↑	F↑	C	E	G	A↑	:	6 2203
C	G	D↑	G↑	D	G	C	D↑	:	5 2211
C	A	D↑	G↑	C↑	F	A	C	:	6 2211
C	G↑	D↑	A	D	G	C	D↑	:	6 2212
C	A	E	A↑	D↑	G↑	C	E	:	6 2212

C	A	F	A↑	E	A	D	F	:	6 2212
C	G	D↑	G↑	C↑	F↑	B	D↑	:	7 2213
C	G	D	G	C	F	G↑	B	:	6 2213
C	G	D	G	C↑	F	A↑	C↑	:	6 2213
C	G↑	E	A	D	G	C	D↑	:	7 2213
C	A	F	A↑	D↑	G↑	C	E	:	7 2213
C	G↑	D↑	G↑	C↑	F↑	B	D	:	7 2214
C	A	F	A↑	D↑	G↑	B	D↑	:	7 2214
C	A	E	A↑	D↑	G	A↑	D	:	7 2214
C	A	E	A	D↑	G	A↑	D	:	7 2214
C	G	D	G	C	E	G↑	B	:	6 2221
C	G	D↑	G↑	D	G	C	E	:	6 2221
C	G↑	D↑	A	D	F↑	B	D↑	:	7 2221
C	A	D↑	A	D	F↑	A↑	D	:	6 2221
C	A	F	B	E	A	C↑	F	:	6 2221
C	G	D↑	G↑	D	F↑	A	C	:	7 2222
C	G↑	D↑	A	D	G	C	E	:	7 2222
C	A	F	A↑	E	A	D	F↑	:	7 2222
C	G	C↑	F↑	B	E	A	C↑	:	7 2223
C	G	D↑	A	D	G	A↑	C↑	:	7 2223
C	G↑	D	G↑	C↑	F↑	B	D↑	:	7 2223
C	G↑	D	G↑	C↑	F↑	B	D	:	6 2223
C	A	D↑	A	D	G	A↑	D	:	6 2223
C	G↑	D	G	C	F	A↑	C↑	:	7 2224
C	G↑	D	G↑	C↑	F	A↑	C↑	:	6 2231
C	G	D	G	C↑	F	A	C↑	:	6 2232
C	G↑	E	A	D	F↑	B	D↑	:	8 2232
C	G↑	D↑	A	D	F↑	A↑	D	:	7 2232
C	G↑	D	G	C↑	F	A↑	D	:	7 2232
C	G↑	D	G↑	C↑	F↑	A↑	C↑	:	6 2233
C	A	F	A↑	D↑	G	B	D	:	8 2233
C	A	E	A	D↑	G↑	C↑	E	:	6 2301
C	A	F	A↑	E	A	C↑	E	:	6 2301
C	A	F	A↑	E	G↑	C↑	E	:	7 2302
C	G	C↑	F↑	B	E	G	B	:	6 2303
C	G	C↑	F↑	B	E	G	A↑	:	7 2303
C	G	D↑	G↑	D	G	B	D↑	:	6 2310
C	G	D↑	G↑	D	F↑	B	D↑	:	7 2311
C	G	C↑	F↑	C	E	G↑	C	:	6 2311

C	A	E	A	D↑	G↑	C↑	F	:	7	2311
C	G	C↑	F↑	B	D↑	G	B	:	6	2312
C	G	C↑	F↑	B	D↑	G	A↑	:	7	2312
C	G↑	D	G	C↑	F	G↑	C	:	6	312
C	G↑	D	G	C↑	F	G↑	B	:	7	312
C	A	E	A↑	D↑	G	B	D↑	:	7	2312
C	A	D↑	G↑	D	F↑	A	C↑	:	7	2312
C	G	C↑	F↑	B	E	A	C	:	7	2313
C	G	C↑	F↑	B	E	G↑	B	:	7	2313
C	A	F	B	E	G↑	C↑	F	:	7	2320
C	G	D	G	C↑	F↑	A↑	D	:	6	2321
C	G↑	D↑	A	D	G	B	D↑	:	7	2321
C	G	C↑	F↑	B	D↑	G↑	B	:	7	2322
C	G	D↑	A	D	F↑	A↑	C↑	:	8	2322
C	G	D↑	G↑	D	G	A↑	C↑	:	7	2322
C	G↑	E	A	D	G	B	D↑	:	8	2322
C	G↑	D	G	C↑	F	A↑	C↑	:	7	2322
C	G↑	D	G	C↑	F	A	C	:	7	2322
C	G	C↑	F↑	C	F	A↑	D	:	7	2323
C	G	D↑	G↑	D	F↑	A↑	C↑	:	8	2323
C	G↑	E	A	D↑	G	A↑	D	:	8	2323
C	A	D↑	G↑	C↑	F↑	B	D	:	8	2323
C	A	F	A↑	D↑	G↑	B	D	:	8	2323
C	A	E	A↑	D↑	G	B	D	:	8	2323
C	A	D↑	G↑	D	F↑	A↑	C↑	:	8	2323
C	A	F	A↑	E	G↑	B	D	:	8	2323
C	G↑	D↑	A	D	G	A↑	D	:	7	2332
C	G↑	D	G	C↑	F↑	A↑	D	:	7	2332
C	G	C↑	F↑	C	E	G	B	:	6	2402
C	G	D	G	C↑	F↑	A↑	C↑	:	6	2411
C	G	C↑	F↑	C	F	G↑	C	:	6	2411
C	G	C↑	F↑	C	E	G↑	B	:	7	2412
C	A	E	A↑	D↑	G↑	B	D↑	:	7	2412
C	A	F	A↑	E	G↑	B	D↑	:	8	2413
C	G	D	G	C↑	F↑	B	D↑	:	7	2421
C	G	D	G	C↑	F↑	B	D	:	6	2421
C	G↑	D	G	C↑	F↑	B	D↑	:	8	2422
C	G↑	D	G	C↑	F↑	B	D	:	7	2422
C	G↑	D	G	C↑	F↑	A↑	C↑	:	7	2422
C	G↑	D	G	C↑	F↑	A	C	:	7	2422

C	A	E	A↑	D↑	G↑	B	D	:	8	2423
C	A	D↑	G↑	D	G	A↑	C↑	:	8	2423
C	G	D	G	C↑	F↑	A	C↑	:	6	2430
C	G	C↑	F↑	C	F	G↑	B	:	7	2512
C	A	F	A↑	D↑	G	A↑	C↑	:	7	3015
C	G	D↑	G↑	C↑	F	A↑	C↑	:	7	3024
C	G↑	D↑	G↑	C↑	F↑	A	C	:	6	3103
C	A	D↑	G↑	C↑	F↑	A	C↑	:	6	3103
C	G	C↑	G	C	F	A↑	C↑	:	5	3104
C	A	E	A↑	D↑	G	A↑	C↑	:	7	3104
C	A	E	A	D↑	G	A↑	C↑	:	7	3104
C	G↑	D↑	G↑	C↑	F	A	C	:	6	3111
C	G	D↑	G↑	C↑	F	G↑	C	:	6	3112
C	A	D↑	G↑	C↑	F	A	C↑	:	6	3112
C	G	D↑	G↑	C↑	F	G↑	B	:	7	3113
C	G	D↑	G↑	C↑	F↑	A↑	C↑	:	7	3114
C	A	D↑	G↑	C↑	F↑	A↑	C↑	:	7	3114
C	A	E	A	D	G	A↑	C↑	:	7	3114
C	G	C↑	G	C	E	A	C↑	:	5	3120
C	G	D↑	G↑	C↑	F	A	C	:	7	3122
C	G	C↑	G	C	F	A	C↑	:	5	3122
C	G↑	E	A	D	F↑	A	C↑	:	7	3122
C	A	E	A	D	F↑	A↑	C↑	:	7	3122
C	A	D↑	G↑	C↑	F	A↑	C↑	:	7	3123
C	G	D	G↑	C↑	F	A↑	D	:	7	3132
C	G	D↑	G↑	C↑	F	A↑	D	:	8	3133
C	G↑	E	A	D	F↑	A↑	C↑	:	8	3133
C	G	C↑	G	C	E	G↑	C	:	5	3200
C	G↑	E	A	D↑	G↑	C↑	E	:	6	3201
C	A	F	A↑	E	A	C↑	F	:	6	3201
C	G	C↑	G	C	F	G↑	C	:	5	3202
C	A	F	A↑	E	G↑	C↑	F	:	7	3202
C	A	E	A↑	D↑	G↑	C↑	E	:	7	3203

C	A	F	A↑	D↑	G↑	C↑	E	:	8	3204
C	G	C↑	F↑	C	E	A	C↑	:	6	3211
C	G↑	D	G↑	C↑	F	A	C	:	6	3211
C	G↑	E	A	D↑	G↑	C↑	F	:	7	3211
C	G	D↑	G↑	C↑	F↑	A	C	:	7	3212
C	G	D	G↑	C↑	F	G↑	C	:	6	3212
C	G	D	G↑	C↑	F	G↑	B	:	7	3212
C	G↑	D↑	A	D	F↑	A	C↑	:	7	3212
C	A	D↑	A	D	F↑	A↑	C↑	:	7	3212
C	G	D	G	C↑	F	G↑	C	:	6	3213
C	G	D	G	C↑	F	G↑	B	:	7	3213
C	G↑	D	G↑	C↑	F↑	A	C	:	6	3213
C	G↑	D↑	G↑	D	F↑	A	C↑	:	7	3213
C	G↑	E	A	D↑	G	A↑	C↑	:	8	3213
C	A	E	A↑	D↑	G↑	C↑	F	:	8	3213
C	A	D↑	A	D	G	A↑	C↑	:	7	3214
C	G	D	G↑	C↑	F	A↑	C↑	:	7	3222
C	G	D	G↑	C↑	F	A	C	:	7	3222
C	G↑	D	G	C	F	A	C↑	:	7	3222
C	G	D↑	G↑	C↑	F↑	A↑	D	:	8	3223
C	G↑	E	A	D	G	A↑	C↑	:	8	3223
C	G↑	D↑	A	D	F↑	A↑	C↑	:	8	3223
C	A	D↑	G↑	C↑	F↑	A↑	D	:	8	3223
C	G↑	D	G	C	E	A	C↑	:	7	3231
C	G	D	G↑	C↑	F↑	A↑	D	:	7	3232
C	A	D↑	G↑	C↑	F	A↑	D	:	8	3232
C	G	C↑	G	C	E	G↑	B	:	6	3301
C	G	C↑	G	C	F	G↑	B	:	6	3303
C	G	C↑	F↑	C	F	A↑	C↑	:	6	3303
C	G	C↑	F↑	C	F	A	C↑	:	6	3311
C	G	C↑	F↑	B	E	G↑	C	:	7	3312
C	G	D↑	G↑	C↑	F↑	B	D	:	8	3313
C	G	C↑	F↑	B	D↑	G↑	C	:	7	3321
C	G↑	D	G	C↑	F	A	C↑	:	7	3321
C	G	D	G↑	C↑	F↑	B	D↑	:	8	3322
C	G	D	G↑	C↑	F↑	B	D	:	7	3322
C	G	D	G↑	C↑	F↑	A↑	C↑	:	7	3322
C	G	D	G↑	C↑	F↑	A	C	:	7	3322

C	G	D↑	G↑	D	F↑	A	C↑	:	8	3322
C	G↑	D↑	A	D	G	A↑	C↑	:	8	3323
C	G↑	D	G	C↑	F↑	A	C↑	:	7	3421
C	G↑	D↑	G↑	C↑	F↑	A	C↑	:	6	4004
C	G↑	D↑	G↑	C↑	F	A	C↑	:	6	4012
C	G	D↑	G↑	C↑	F	A	C↑	:	7	4023
C	G	D↑	G↑	C↑	F↑	A	C↑	:	7	4113
C	G↑	D	G↑	C↑	F	A	C↑	:	6	4210
C	G↑	D	G↑	C↑	F↑	A	C↑	:	6	4212
C	G	D	G↑	C↑	F	A	C↑	:	7	4221
C	G	D	G↑	C↑	F↑	A	C↑	:	7	4321

9 Melody

Melody holds a special place in the hearts and minds of people in most cultures—so special that what follows will arouse as much controversy as any commentary on politics or religion. But what follows is not exactly opinion; it is a demonstration and explanation of a hypothesis. In the arts, as in the sciences, hypotheses lie behind all creative work. Also, in both fields, the value of a particular hypothesis depends on its being self-consistent and not on its being "true." In this spirit, I intend to explain and demonstrate a hypothesis on melody and not to pontificate as to the "truth" about melody.

Melody is as much a succession of pitches in time as is tapestry a sequence of fibers in space. Patterns of temporal and harmonic form are required to weave pitch strings into an appealing fabric. Serious composers seek patterns that are complex or subtle to some degree beyond the currently acceptable norm. That mean level of acceptability is embedded in the popular or folk music of a culture. In our society, a number of subcultures coexist, and economic pressures promote certain styles more than do esthetic pressures. To write in any one style, a composer must either know precisely which patterns are appropriate, or else be so limited in experience that only those stylistic patterns exist in his musical vocabulary. What an individual considers "meaningful" in any component of an artform, or the way his scales of preference rank the various styles, depends entirely on his experience with the patterns involved. But more later on why this is so.

Restricting our scope to the MT idiom does not create a firm base for esthetic reference. A melodic event of charming interest to one listener may seem insipid to another. A different event will engage the interest of the latter listener but appear chaotic to the former. For listeners at one end of the spectrum of experience, music and melody are practically synonymous—assuming that melody (and not just the lyric) is "heard" at all—and, if melody is to be

recognized as "melody," it must be highly "tonal."

TONALITY

Tonality is a property of music in which a key or "flavor of a scale" is obvious. At the avant-garde end of the listeners' spectrum, tonality is a synonym for banality. To approach a middle ground where no listener should be too dissatisfied with the demonstrated melodies, I will use tonality as the key to compromise. In the most traditional sense, *tonality* is an enduring property of a melody, such that a listener never loses the tonality of a strain even though any number of chord changes may occur.

Over time, tradition has yielded to experience by allowing *modulation* to take a melody from one key to another and still be perceived as tonal. Frequently such modulation takes place only in the harmony (the chords that "harmonize" the melody); the key signature of the piece does not necessarily change.

Still later developments extend this trend by eliminating key signatures altogether and decreasing the length of time during which one "key" must exist. This makes tonality a property of simultaneous collections of pitches. Such a collection—a chord and the melodic tones it harmonizes—need only have a recognizable "flavor." I will not limit such flavors to the traditional ones such as major or minor. In later chapters, I'll clarify the idea of flavor by quantifying the concept of tension.

While such ideas may strike the nonmusician as totally abstract and detached from his everyday experience with melody, it is a fact that he would find it difficult to recognize his favorite melodies without sensing the presence of their "correct" tonalities. Tonalities are structures of pitches, but I want to concentrate more on that part of my hypothesis that claims "sensibility" in melody also depends on a kind of internal temporal structure that can be perceived, at least subconsciously. This, too, is quantifiable, but structure of any sort can only be adequately described in terms of the geometry of the space in which it exists and the materials and techniques available for constructs in that space.

MELODIC SPACE

Melody in the abstract, without considerations of instrumentation, dynamics, or any other nuances associated with performance, does exist in the two-dimensional realm of pitch and time. As is true for other dimensions, these have no inherent structure until we impose reference grids on them (Fig. 9-1). Unlike our treatment of other dimensions, we tend to assign reverence rather than mere reference to these lines. The horizontal ones do more than measure pitch, they represent the "allowed" tones in our pitch system or possibly in our chosen scale within that system. The vertical lines represent convenient points in time corresponding to a given tempo, and all "allowed" rhythms must fit rationally (in the sense of rational numbers) on and between these metric markers.

So we see the space of melody is quantized or given a microstructure of its own right from the start. This microstructure then serves as a stage. Our melodies must have a higher level of structure in which, like actors appearing on this stage, melodic events will be directly observable from moment to moment. In a "melodic event," components of "motion" along the pitch axis establish and develop prevailing harmonies while temporal components of motion do the same for rhythm. Melodies also need a still higher level or macrostructure in which events are ordered and interrelated as in a scenario. Contiguous events may show contrast, or they may exhibit relationships to, or evolution from, each other. Carrying the theatrical metaphor to a comparison of "Sesame Street" and "Masterpiece Theater," one's awareness at the scenario level reflects one's experience and maturity in contemplating and remembering patterns in this particular "space." Though the listener's mind need only be concerned with the patterns, the composer, like the playwright, needs to be aware of the properties of the

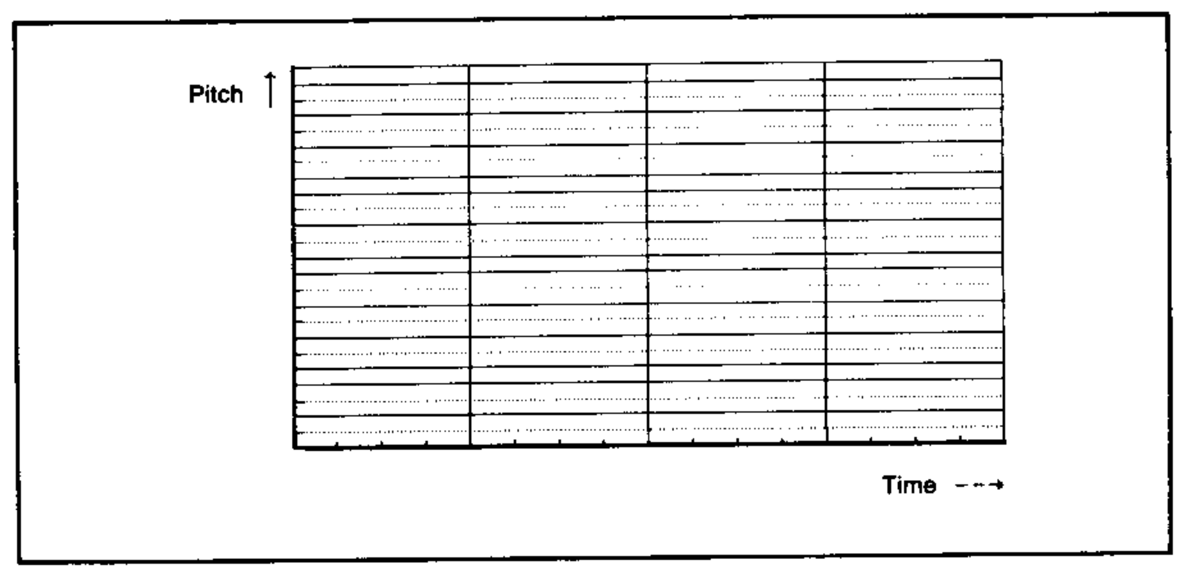


Fig. 9-1. A reference grid for pitch-time space.

"space" containing the patterns!

The pitch and time microgrids seem intuitively clear as long as we think of them as defining the space of the MT idiom. To understand this space, rather than just be aware of it, we need to consider it with reference to more general forms of music as well as to our sense of hearing.

TUNING SYSTEMS

Multimedia artists working with computers frequently assume that the microgrid for pitch must have some natural structure that we can sense, or that the grid usually assigned representing the equal temperament tuning system is itself natural and not at all arbitrary. Another frequent assumption links the sense of hearing with that of vision. Here, different pitches are supposed to be related in some way to specific colors or some other visual entities. There is nothing wrong with such hypotheses as long as they are applied in self-consistent ways. But it is a pity that so much time and effort is spent trying to "prove" them because they contradict the results of physical experiments, not to mention the physical hypotheses that are consistent with those experiments. The problem here is that the artists involved confuse the patterns they sense with the medium or space in which the patterns appear.

The physical mechanisms that produce and detect pitch are very different from those that generate or respond to light. When the air around us carries oscillations roughly in the range between 20 and 20,000 vibrations per second, our aural sense creates in our minds the sensation we call sound. When electromagnetic oscillations in an unimaginably high frequency range (hundreds of trillions of vibrations per second) reach our eyes, our minds construct a world containing colors, shapes, textures, and so on. If any connection exists between the two senses, it, too, is all in the mind. Our minds beguile us, telling us that its constructs are fundamental. Even this idea that claims "objective reality" is all in the mind is itself a product of the mind. So feel free to hypothesize as you like, but remember: Whenever your ideas contradict what can be tested, compositions based on them will probably not seem "sensible" to others no matter how "correct" they seem to you.

There is one aspect to our tuning system that might be called "natural." There is a subtle relation between the nature of so-called harmonic oscillators and the structure and response of the

normal hearing mechanism. The lack of interference between tones whose frequencies differ by a factor of two has a universal sensory effect and has given many musical cultures a framework on which to build tuning systems. Consider the frequency values:

110 220 440 880 1760 . . .

While 1760 is twice as far from 880 as the latter is from 440, we find it natural to "sense logarithmically" and call each separation one octave. Something within us claims each octave separation is equal in magnitude. Yet, different cultures have subdivided the octave differently to form their pitch scales.

The system considered standard in Western music today has a precise mathematical formulation—though in practice, such precision is intentionally avoided. It is based on raising the number two to a rational power using $2*M \div 12$, where M is an integer. Using A-440 as the reference frequency, $440 \times 2*M \div 12$ gives the frequencies of all pitches in the equal temperament system. For M equal to three, this gives the frequency of the note three semitones above the reference pitch (C). For a value of negative three, the computed frequency corresponds to the F# three semitones below A. Notice that when M equals zero, the expression reduces to 440×1, and for M equal to twelve, it becomes 440×2; setting M to integer multiples of twelve gives the frequency of A in different octaves.

Using computers to generate sound, it is easy to experiment with equally tempered tunings that use different numbers of pitches in the octave. Changing the 12 to a 24 keeps the conventional system intact (in that all even values for M reduce the fraction) and superimposes a similar system "between the cracks" (odd values for M) to produce quarter tones. For a sixth-tone system, you would use 36 in place of 12, keeping the current system but adding two "equally spread" sets of such tones between the conventional ones. Of course, you can also ignore convention and replace the 12 with 11 or 17 or 21 or whatever you like. But I'd like to suggest an evolutionary approach that would not lose the ordinary listener in totally unfamiliar surroundings and would not lose the composer in totally unfamiliar systems of harmony.

The tones of various chord structures can be taken as fixed reference points in a tuning. This takes advantage of our powerfully ingrained harmonic sense. For example, suppose the notes C, E, and G are fixed. Now between C and E, instead of the three conventional semitones, let's introduce just two equally tempered notes using $2*M \div 9$ as the tuning factor. That is, calling FC the frequency of the reference pitch, C,

$$FC \times 2 * 0 \div 9 = FC$$
$$FC \times 2 * 1 \div 9$$
$$FC \times 2 * 2 \div 9$$
$$FC \times 2 * 3 \div 9 = FE$$

(Note: $2*3 \div 9 = 2*1 \div 3 = 2*4 \div 12 = FE$, where FE is the frequency of the pitch, E.) Similarly, between E and G we might interpose three intermediate tones to replace the two conventional ones by breaking the minor third ($2*3 \div 12$) into four parts:

$$FE \times 2 * (3 \div 12) \times 0 \div 4 = FE$$
$$FE \times 2 * (3 \div 12) \times 1 \div 4$$
$$FE \times 2 * (3 \div 12) \times 2 \div 4$$
$$FE \times 2 * (3 \div 12) \times 3 \div 4$$
$$FE \times 2 * (3 \div 12) \times 4 \div 4 = FG$$

And from G to the next C, three more pitches could replace the conventional four using:

$$FG \times 2 * (5 \div 12) \times 0 \div 4 = FG$$
$$FG \times 2 * (5 \div 12) \times 1 \div 4$$
$$FG \times 2 * (5 \div 12) \times 2 \div 4$$
$$FG \times 2 * (5 \div 12) \times 3 \div 4$$
$$FG \times 2 * (5 \div 12) \times 4 \div 4 = 2 \times FC$$

We would now have an eleven-tone system based on and preserving the flavor of a particular triad. This could be used as *the* tuning system or as only one tuning in a system where each harmonic structure carries its own pitch space!

To sum up in a general way about the structure of the pitch microspace, we can say $F \times L^{*}M \div N$ creates a system in which the interval between a base frequency F and L times F is broken into N parts. And the aspect of logarithmic tuning that makes it useful is that we respond to those N parts as being equally spaced. This allows a given melody to be transposed so as to begin on any pitch in the system and still be recognized as the same melody. Since there is no reason to argue with our senses, we can then associate the tuned pitches with a linear rather than a logarithmic grid. That is why I drew the horizontal grid lines with equal spacing, and that is why we might just as well label them as C, C♯, D, etc., instead of using numerical values representing unequally spaced frequencies.

THE MICROGRID FOR TIME

In labeling the grid for the time axis, we again face the formidable problem of rhythmic notation that has harassed generations of musicians. Suppose we decide to construct the grid so that vertical lines are an inch apart and each such separation corresponds to one beat, i.e., a quarter note in duration. If we expect the smallest duration we will need to be a sixteenth note, we could add subsidiary grid lines at the half-inch marks to show where eighth notes will be attacked, or even include quarter-inch marks to pinpoint the sixteenth notes. But if some sixteenths are to appear in artificial groupings where five of them must fit into the time of four ordinary ones, four subdivisions per inch will prove as awkward as five. Twenty per inch would allow a neat fit for all notes, but might provoke a fit for your optic nerves. And should we decide to include eighth-note triplets, we could request 60 subdivisions per inch and a microscope.

Clearly, our ears are more expert at dividing time than pitch. If they were equally proficient in both areas, we would be able to divide the octave as readily as we can divide the beat and so make use of different tuning systems simultaneously and in sequence. We should not then expect to be able to give each "allowed" pitch its own grid line. But our musical sense has not evolved to that point. Speech has given us such facility with rhythm, so that the vertical lines on our grid need not legibly mark every allowed attack time. A convenient unit, such as an inch per beat, is adequate for representing the temporal grid. The positions of "notes" could be approximated visually against such a background. But attacks and durations can be expressed mathematically to any required degree of precision, and we can "see," whether they are drawn on paper or not, how they must fit into the time dimension.

HIGHER LEVELS OF STRUCTURE: DESIGN VERSUS ACCIDENT

With a comfortable degree of understanding and control over the elements of musical space, we can now turn our attention to the higher levels of melodic structure. Sensibility in melody derives from the fitting together of melodic fragments so as to create a larger pattern of tones which seems to be a logical result of the generating fragments. The overall shape of the larger pattern needs to satisfy certain expectations based on experience. This is not unlike sensibility in a novel or even in a building. Imagine the gripping suspense of a story line in which randomly selected characters appear in a random continuity of events, or the magnificance of a structure engineered by placing whatever piece of hardware happens to be next within reach into a position chosen without regard for function, geometry, or gravity. In each case there is some chance that the result will have appeal—but it would be the appeal of levity based on the pseudosensibility of nonsense!

Significant new ideas formed in the sciences have a way of filtering down to the common awareness and triggering responses in the arts. The role of randomness in quantum physics is such an idea, and its misapplication in the arts will provide future historians with at least one humorous anecdote of this century. Over the past few decades, serious artists and musicians as well as dabblers have tried throwing paints at a canvas and notes at a score. Because of the cohesive forces between

paint molecules, the artist can count on some essence of "structure" in his results. The musician has no such guarantee. He places himself in the position of the alchemist, believing that order in method will substitute for order in the materials being used. In the Middle Ages it made sense to suppose that if you mixed enough different substances over a long enough period of time, sooner or later you were bound to change lead into gold. I say "it made sense" because, when you don't know what to do and the necessary knowledge isn't even available, you might just as well try anything. But in an age when the difference between substances, elements, and elementary particles is recognized, it no longer makes sense.

Today there is also knowledge available from the fields of thermodynamics or information theory that point out the folly of attempting to transmute the lead of randomness into the gold of melody. The important conclusions and their pertinence to the arts can be briefly summarized. Left to themselves, things tend toward increasing randomness. To achieve or maintain order, effort must be expended. As a result, through experience, order is interpreted by our minds as the imprint of intelligence and creativity. Our minds constantly look for ordered patterns in the information collected by our sensory organs. If you understand the patterns required and take the trouble to compute the probability of a melody being produced by the effortless chances of chance alone, you'll find that, for all practical purposes, plain chance will never even produce plain chants. What is it, then, that constitutes order in melodic patterns, and how can it be designed in a rational way?

BUILDING BLOCKS OF MELODY

In treating rhythm, we saw how style depended on developing microrhythms into macrorhythms. Melody depends on exactly the same sort of development, but there are micropatterns in pitch, as well as time, to be considered. A pitch micropattern might be as uncomplicated as a single tone or a single interval. As with rhythm, putting more structure into the basic building blocks assures more character in results that are based on simple developmental schemes such as repetition and permutation. For purposes of demonstration, I want to introduce a more involved sort of micropattern, one based on logic, and yet keep things very simple.

Let's define a thematic motif that is independent of rhythm, so that we concentrate only on the patterns of pitch being developed. Without even limiting the number of pitches that can be contained in the micropattern, although we know it must be "small," we might ensure cohesiveness based on repetition by saying that every other note, beginning with the first, must be the same. As an afterthought, to force the motif to end on the repeated pitch, let's say there must be an odd number of notes in the micropattern.

Now we need an algorithm that will convert this abstract motif into observable melodies. I would also like to limit the amount of abstraction in the description, so let's use specific notes or numbers and preserve our thoughts, as they develop from this point on, in APL notation. The simplest odd number I can think of (that will allow the first pitch to be repeated) is three. So we'll initialize a string of three notes (0, 0, 0) where zero will imply the pitch, C, by writing:

```
P←3ρ0
```

According to the definition of the pattern, the second pitch here, which has an index of one, can be altered. Keeping things painfully simple, let's do this by adding one to it:

```
P[1]←P[1] + 1
```

Had we chosen a longer motif, we could also change P[3 5 7 . . .].

At any rate, we now have a single musical brick and must decide what to do with it. Don't be misled by the specificity here; any number of other definitions could have been used to produce a fundamental pattern with or without including rhythm in the logic.

REPETITION: TRANSLATION AND TRANSPOSITION

The most obvious and constructive thing to do with a brick, even if it only exists in time, is duplicate it and place the two end-to-end. In APL terminology, the beginnings of a melody named M can be formed by catenating the three-pitch motif P with itself:

M ← P, P

Notice that a rhythm is repeated by translating it along the time axis. In the catenation of the pitch motif here, we will sense the repetition in time so that a pattern of pitches is also repeated by translation along the *time* axis. But a pitch pattern can also be repeated with translation along the *pitch* axis by adding a number to it. A musician would say the pattern P was *transposed* up a minor third by the expression **P + 3**. Patterning operations that involve transposition can be rather complex, but they are so useful that, over time, one tends to collect them in the form of subroutines with parametric control. Suppose, for example, we want a routine that will add a given interval or sequence of intervals, I, to the first N or last (minus) N pitches of the global melody, M. I just happen to have on hand a function I invoke by thinking, "N Take Melody Plus I" (Fig. 9-2).

Using this, we can combine repetition (in time) of all or part of the melody with transposition (repetition in pitch). There are possibilities inherent in this generalized form of "repetition" that show no obvious trace of repetition in the normal sense. By writing

M ← M, ¯1 TMP 1

we extend the melody by a single note, appending one pitch that is derived from the last note (the minus one here) of M by adding the value (plus) one to it. So this algorithm for repetition can also generate intervals or pitches to connect or extend patterns. Here, the added pitch will extend the repeated motif and also serve as a connector for what follows. If we repeat this entire M-vector, that pitch will seem part of a larger pattern.

As I said, we could use this subroutine to repeat the entire melody transposed through a given interval. But there are times when you want the repetition to begin on a pitch that can be described logically or specifically, but you can't express it as an interval. Figure 9-3 is a "one-liner" that will transpose the melodic vector V to BeGiN on the pitch S (for Start).

With control over pitch as well as interval, we can now repeat the entire melodic string transposed through some given interval from some given pitch! The expression

M ← M, M BGN ¯1 TMP 1

adds to the entire melody (**M,**) the entire melody (**M**) transposed to begin (**BGN**) on the note that is one semitone higher than the last note of the untransposed melody (**¯1 TMP 1** again). Repetition of the use of the **TMP** function with identical parameters certainly makes it appear to be a legitimate part of the patterning in a logical sense. It would be nice to believe it also does so in a musical sense, but this demonstration must be based on reason, not faith.

The choices for the various parameters certainly need not be so uninspired. But I'm trying to demonstrate "pattern" as if it could be divorced

```
      ∇ Z←N TMP I;K
[1]     K←⌈/(ρ,I),|N
[2]     Z←(KρI)+KρN↑M
      ∇
```

Fig. 9-2. Function TMP transposes N pitches in melody M through the interval(s), I.

```
      ∇ Z←V BGN S
[1]     Z←(V−V[0])+S
      ∇
```

Fig. 9-3. Function BGN transposes the notes in V so that they begin on pitch S.

from the material it contains, and I don't want that material to mask the method or to imply any musical preconception of the results.

DISTORTIONS

So far, we have used simple repetition and repetition combined with transposition to develop a melody from a motif. In introducing the TMP function, I got carried away and inserted a stray, but related, pitch which might be classified as a distortion, an extension, a connector, or a blunder, depending on your outlook and the final result. A more contrived distortion should take advantage of the distortable characteristics of the *logic* behind the basic building block.

Remember, the logical description of our motif is far more general than the current setting of the P-vector would suggest. In terms of that general description, a form of distortion comes to mind in which the repeated pitches are left unaltered, but the "in-between" pitches have successively larger intervals added to them. That is to say, given **P[0 1 2 3 . . .]**, where all the even-indexed notes are the same, we could add the interval vector 0 ¯1 0 ¯2 0 ¯3 0 ¯4 . . ., so that the tones with even indices would still all be the same while the odd-indexed pitches would be lowered through increasing intervals. I used negative numbers here to show that "increasing" intervals are not necessarily positive; the successive nonzero intervals could certainly all be positive or even be made to spiral outward by alternating their signs.

In the particular case at hand, the motif has only three notes. For clarity, I'm going to use only the first three terms of the distortion in the following expression:

M ← M, P ← (P BGN ¯1 TMP 1) + 0 ¯1 0

Besides appending the distorted pitch motif to the melody, the original motif, P, is purposely being reset here (**P←**) to avoid the "partridge-in-a-pear-tree" syndrome in further development. P will now contain the original motif transposed so as to begin one semitone above the last note of the undistorted melody and be itself distorted by the interval sequence 0 ¯1 0. When we get around to generalizing these statements, those three numbers will need to be replaced by something like

((ρP) ρ 0 1) \ −1 + ι P ≠ 0.

Another kind of distortion can use insertion, deletion, or replacement. Because replacement combines insertion and deletion, let's substitute three or four pitches for the next-to-last pitch in a repetition of the last sequence, the "new" P. We can make the inserted pitches "dance" around the featured (repeated) pitch of the motif with a step Arthur Murray might diagram as the upper part of Fig. 9-4.

The circled X marks the starting and ending point. Musically, let's say the large paces take us three semitones away from the target pitch. In this idiom, we must not step on the cracks between pitches, so the smaller intervals cannot all be equal. Of course, they must be integers that add appropriately to three, so we'll set them up as shown in the lower part of the figure.

If you recall that the more traditional scales are built of the intervals one and two, it will be apparent that my choices of the values one, two, and three here were not completely arbitrary. The inserted pattern should now appear to carry a scale-like tonality! Specifically, we can write:

M ← M, (P[0] + 0 3 1 ¯3 ¯2), P[0]

This takes advantage of the fact that, with only three notes in the current P-vector, the next-to-last pitch is the only one not necessarily equal to **P[0]**. In generalizing for a longer motif, we will probably use the form:

M ← M, (¯2 ↓ P), (P[0] + 3 1 ¯3 ¯2), P[0]

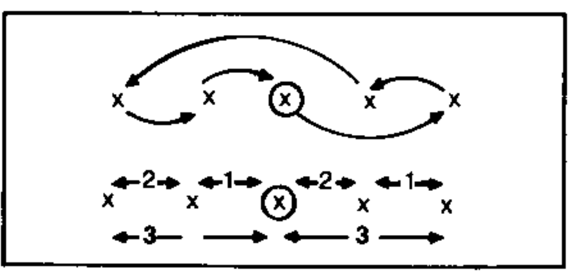


Fig. 9-4. "Dance-step" diagram showing the movement of inserted pitches around a featured pitch. The interval structure for the "steps" is shown below.

where the term, (¯2 ↓ P), catenates to M a repetition of the entire phrase except for the last two notes, and the replacement tones follow along with the final P[0].

CLIMAX

Consider how the melody so far developed might be expressed verbally. It says, "Here is the basic pattern of pitch. Here it is again. Just to make sure it's clear, here it is two more times, transposed but connected to the previous statement by an additional tone. Now, with the same kind of connecting tone, here it is again with a twist, and still once more with a more complex distortion."

Now, it must get to the point of its monologue with a logical conclusion. Conclusions, whether of a phrase or an entire piece, need to contain particular kinds of patterns that I can only describe in terms of the patterns we experience in the physical world. Our melody needs an ending that follows two increasingly distorted statements of the thematic motif. Distortions lead in unexpected, and therefore interesting, directions. But the musical vehicle should take the listener to an inviting destination, moving and coming to a stop in a "natural" way—i.e., without violating the laws of motion that we all have come to understand through experience. In fact, our goal is to "move" the listener emotionally, not physically; only the melody is in motion, and that is the motion that must appear to be "natural."

Our thematic material directs the listener's attention to a repeated pitch. Even though the motif undergoes transposition and distortion, that repeated pitch becomes and remains the most prominent member of the total pattern. It as though most of the "momentum" resides in that pitch, and we must now decide how to bring the pitch to rest in a convincing way. A sudden stop at this point would have all the persuasive realism of an archer's arrow looping several times around the target and then dancing hesitantly just before its final thud into the target. No, our motion so far has been more like that of an athlete getting into position to do his thing. In analogy to a pitcher stepping up to the mound or a diver launching himself toward the board's edge, let's consider the last note as a high point where all the accumulated energy of motion has been focused and must now be released.

Considering just the pitch and time dimensions of melodic space, our experiences in three-dimensional space and time with the force of gravity and the inertial qualities of matter carry over in surprisingly obvious ways. That is, we associate heavy, cumbersome motion with low pitch moving slowly, and light, agile movement with high pitch in rapid motion. Up and down translate directly into high and low in the pitch direction—but so do back and forth, because we have no control over "direction" in time. But the structure of ordinary space and motion in that space appear continuous, while our pitch space is quantized and our pitch motions are tonal, so that consecutive notes in the melody must be separated by scale-like intervals! As a result, the patterns of sensible, smooth motion in ordinary space must transcribe directly into scale-like patterns.

A scale-like pattern is not necessarily the same as a scalewise pattern. To be scale-like, all the notes need only be members of a scale with quasi-tonal properties. Such scales usually contain only the intervals one and two between adjacent tones, and the latter interval predominates, logically enough, by about two to one. So, let's construct a pattern that will convincingly "home in" on the featured pitch P[0]. The simplest such pattern I can envision would use P[0] as the high point of a repeated ascending motif. A general algorithm for creating the ascending sequence would depend on three arguments: the interval structure of the scale, the target pitch, and the number of tones in the sequence. A special algorithm could assume a particular interval structure, say 2 1 2, as in Fig. 9-5.

The left argument here is the target tone. The right argument is a positive number that will be increased by two to establish the number of tones in the sequence. (This indirect approach helps to assure that the sequence will indeed be a sequence by containing at least three notes. It assumed I was the only user and I could at least remember that N must be a positive integer. For wider use, it really

```
     ∇ Z←T CDNC212 N;K
[1]    Z←+\(T−+/K),K←(2+N)⍴ 2 1 2
```

Fig. 9-5. Function CDNC212 approaches target pitch T from below in N + 2 steps, repeating the interval sequence 2 1 2 as needed.

should ask for specific input and test the given values.)

Now if we execute

P ← P[0] CDNC212 1

the new P will contain three (two plus the given argument, one) successively higher notes leading up to the target tone, **P[0]**. To give this passage the energy focusing qualities of a closing cadence, we might employ considerable ingenuity using permutations or any other sort of development. Or, we can simply submit it in triplicate:

M ← M, P, P, P

and thereby assume it will be duly noticed.

This repeated pattern could bring to mind the energetic properties of a pitcher's windup. If what follows is to be limited to a comfortable range for voice, the release should not suggest a wild pitch. Without examining the results to this point, let's allow the melody to move past the target pitch by only one semitone and then drop through the lower end of the repeated phrase P by the same amount:

M ← M, (1 + ¯1 ↑ P), ¯1 + 1 ↑ P

Fortunately, should our conservative precautions prove inadequate to keep the pitch in Singer Stadium, there is such a thing as instrumental music.

GENERALIZATION

A listing of the complete terminal session appears in Fig. 9-6, showing also the final form of the melody as a numeric vector and, through the use of the **N2P** function, as a pitch string in alphanumeric notation. Figure 9-7 represents the result as it appears in musical notation. Much as I would like to dwell on the inspirational qualities of this result, there are some less musical matters that command attention first.

Though the entire procedure contains about only ten lines of code, most of that is "pure algorithm" with little variable data. Testing the

```
      P←3⍴0
      P[1]←P[1]+1
      M←P,P
      M←M,¯1 TMP 1
      M←M,M BGN ¯1 TMP 1
      M←M,P←(P BGN ¯1 TMP 1)+ 0 ¯1 0
      M←M,(P[0]+ 3 1 ¯3 ¯2),P[0]
      M←M,P,P,P←P[0] CDNC212 1
      M←M,(1+¯1↑P),¯1+1↑P
      M
0 1 0 0 1 0 1 2 3 2 2 3 2 3 4 4 4 7 5 1 2 4 ¯1 1 2 4 ¯1 1 2 4
      ¯1 1 2 4 5 ¯2
      N2P M
0.  C    C↑   C    C    C↑   C    C↑   D    D↑   D    D    D↑
     D    D↑   E    E    E    G    F    C↑   D    E   ¯1B  +1C↑
      D    E   ¯1B  +1C↑  D    E   ¯1B  +1C↑  D    E    F   ¯1A↑
```

Fig. 9-6. A listing of the terminal session described in the text.
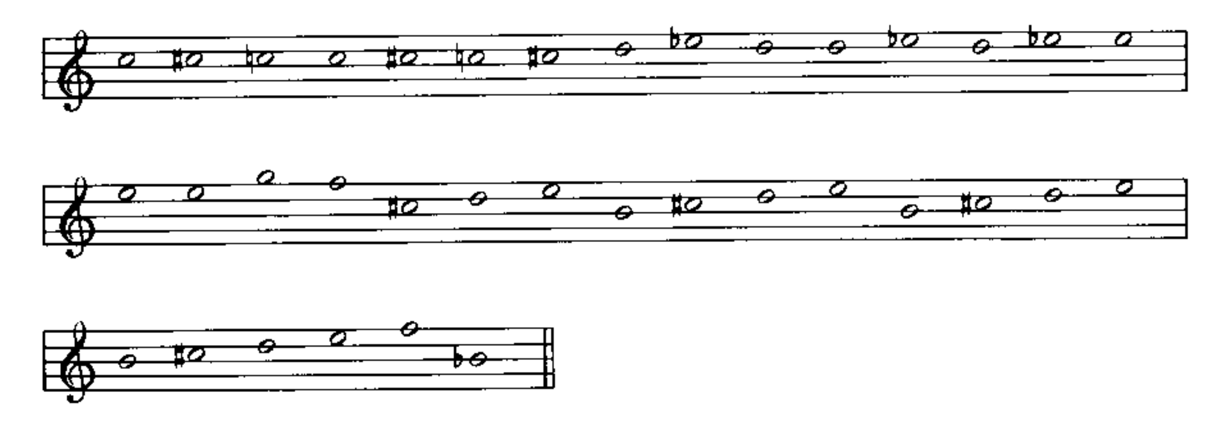


Fig. 9-7. A musical transcription of the results of the terminal session listed in Fig. 9-6.

algorithm with a variety of data would quickly become an onerous chore, even for someone less indolent than I. So one reason for restating the procedure in the form of a generalized function (Fig. 9-8) is to increase its usefulness by reducing the burden associated with examining large numbers of possibilities. In exhibiting such a generalization immediately, I do not mean to infer that the task of creating this function from the previous example is entirely trivial.

In conversion to functional form, wherever numbers appear in the original, one tries to replace them in a way that permits the composer to select values ad lib. The easiest and most direct way to provide freedom of choice is to collect the numerical information in the arguments of the function header. For the sake of clarity I have limited this information to the starting (and repeating) pitch S, and the intervals defining the "in-between" tones, IT. In the original, I glibly chose the value one (1) for some totally unrelated entities—the "odd" pitch interval, the number of "odd" pitches, the interval for a connector pitch added after the last pitch of the repeated motif, the interval from that added pitch for beginning the transposed motif, etc. In the generalization, I have interpreted some of these choices to be rationally connected simply because they were all equal. This explains the use of the number of "odd" pitches (⍴IT) in a number of odd places. The arguments for the **TMP** function and

```
     ∇ S DVLP2 IT;P
[1]    M←P←(1+2×⍴,IT)⍴S
[2]    P[1+2×⍳⍴,IT]←1 TMP IT
[3]    M←P,P
[4]    M←M,¯1 TMP 1+⍳⍴,IT
[5]    M←M,M BGN ¯1 TMP 1
[6]    M←M,P←(P BGN ¯1 TMP 1)+((⍴P)⍴ 0 1)\-1+⍳⍴,IT
[7]    M←M,(¯2↓P),(P[0]+ 3 1 ¯3 ¯2),P[0]
[8]    P←P[0] CDNC212⍴,IT
[9]    M←M,P,P,P
[10]   M←M,(1+¯1↑P),¯1+(⍴,IT)↑P
[11]   N2P M
     ∇
```

Fig. 9-8. Function DVLP2 generalizes the terminal session of Fig. 9-6.

the intervals for the "Arthur Murray dance step" have been left unaltered. Ordinarily, more logic would be used to allow more freedom in all these places. Without it, an obvious imprint will be left on each melody produced by this function. All such melodies will appear as highly stylized variations on a theme if exhibited together, yet the range of variation can be astonishing.

To reproduce the original result in which the starting pitch was zero (C) and there was a single "odd" pitch whose interval from the repeated note was one, we need only type

```
0  DVLP2  1
```

and that result will be printed directly. (At least, the intended result will be so printed. Here is a case where the generalization pointed to a typing error in the original procedure: In the seventh line, in copying the simplified form for the dance step, I omitted the zero! It took one of my rare, sharp-eyed moments to see that the pitch E appears four times in succession in one result and three in the other, and many more agonizing moments to spot the source of the problem.)

As a demonstration of the ease of use and the musical potential, Fig. 9-9 is a page from a terminal session containing five successive executions of the function with arbitrary choices for the starting notes and intervals for the varying or odd-indexed pitches. The fact that the arguments were chosen arbitrarily should not be seen as contradicting my earlier statements about randomness. These values were chosen after all the logic was in place, and that logic assures "ordered" results no matter what values are used.

In our culture, such naked melodies have the appear of a nudist camp in the eyes of Pierre Cardin. Not unlike Mr. Cardin, we must now drape them in becoming rhythms and finely woven harmonies to make them presentable.

RHYTHM AND HARMONY

Just as we are attuned to accept tonal melody as sensible, we have had certain rhythmic requirements drummed into our subconscious sense of meaningful musical metrics. If we were to allow our computer to run random rhythmizations of any given melody, we could not avoid insulting this natural metric sense. We have seen that our speech patterns show many of the requirements for fitting beats to discrete sounds, whether syllables or pitches. In speech, we are most concerned with the way we subdivide the beat. We carry this over to melody with additional requirements on the way beats must fit into a larger pattern of measures and phrases.

Given the total number of notes in a melody, it is far easier to program a rhythmical setting based on attacks than on durations, and this lets us use our highly developed linguistic sense for filling in the details. As a concrete example, our original melody here has 36 pitches (at least, that was the count before I discovered the error). The original building blocks form a pattern of seven pitches (the repeated three-note motif and the added pitch separator). This suggests we try to subdivide the entire melody into five phrases, each containing roughly seven notes—because 5×7 nearly equals 36.

Before fixing the number of attacks in a phrase, a somewhat subjective evaluation of the nature of the pitch pattern needs to be made. If the pattern of seven pitches seems complex, the listener will appreciate it more if it coincides exactly with a phrase, because complexity bears repetition and repetition reduces apparent complexity through familiarization. On the other hand, if the pattern seems simple, repetition would bore the listener, so it would be better to have the metric phrase contain one pitch more or less than the fundamental melodic pattern. In our case I must admit I find the seven-pitch pattern repulsively trite, so I'll create interference between the melody and the meter by using eight pitches in a phrase.

The next step has to do with the interaction of harmony with meter. An infallible (though at times insipid) way to make sensible phrases is to insist they contain four or eight groups (bars or measures) of beats. Remember, the number of beats in such a group defines the meter and, if necessary, each

```
      1 DVLP2 2 3

0:   C↑   D↑   C↑   E    C↑   C↑   D↑   C↑   E    C↑   D    D↑   E
     F↑   E    G    E    E    F↑   E    G    E    F    F↑   G    G↑
     G    G↑   G    G    G↑   G    A↑   G↑   E    F    G    C    D
     D↑   F    G    C    D    D↑   F    G    C    D    D↑   F    G
     G↑ ¯1B  +1C↑

      2 DVLP2 ¯3 4

0:   D  ¯1B  +1D    F↑   D    D  ¯1B  +1D    F↑   D    D↑   E    F
     D    F    A    F    F    D    F    A    F    F↑   G    G↑   E
     G↑   A↑   G↑   G↑   E    G↑   B    A    F    F↑   G↑   C↑   D↑
     E    F↑   G↑   C↑   D↑   E    F↑   G↑   C↑   D↑   E    F↑   G↑
     A    C    D

      4 DVLP2 2 7

0:   E    F↑   E    B    E    E    F↑   E    B    E    F    F↑   G
     A    G  +1D  ¯1G    G    A    G  +1D  ¯1G    G↑   A    A↑   B
     A↑ +1D↑ ¯1A↑   A↑   B    A↑ +1C↑ ¯1B    G    G↑   A↑   D↑   F
     F↑   G↑   A↑   D↑   F    F↑   G↑   A↑   D↑   F    F↑   G↑   A↑
     B    D    E

      8 DVLP2 ¯2 ¯7

0:   G↑   F↑   G↑   C↑   G↑   G↑   F↑   G↑   C↑   G↑   A    A↑   B
     A    B    E    B    B    A    B    E    B  +1C    C↑   D  ¯1B
   +1D  ¯1F  +1D    D  ¯1B  +1D    F    D↑ ¯1B  +1C    D  ¯1G    A
     A↑ +1C    D  ¯1G    A    A↑ +1C    D  ¯1G    A    A↑ +1C    D
     D↑ ¯1F↑   G↑

      5 DVLP2 2 2 1

0:   F    G    F    G    F    F↑   F    F    G    F    G    F    F↑
     F    F↑   G    G↑   A    B    A    B    A    A↑   A    A    B
     A    B    A    A↑   A    A↑   B  +1C    C↑   D    C↑   C↑   C↑
   ¯1B  +1C↑   C↑   D    C↑   C↑   C↑   E    D  ¯1A↑   B  +1C↑ ¯1F
     G    G↑   A↑ +1C    C↑ ¯1F    G    G↑   A↑ +1C    C↑ ¯1F    G
     G↑   A↑ +1C    C↑   D  ¯1E    F↑   G
```

Fig. 9-9. Five consecutive executions of DVLP2 using arbitrary input parameters.

beat can be subdivided into two or more attacks to account for the melodic tones. Harmony must be assigned to "fit" these pitches through considerations which we'll take up shortly. If more than four different pitches appear in succession in a melody, it may be difficult, if not impossible, for a given algorithm to find a chord that harmonizes them all. If only a few unique tones appear repeatedly in a

long sequence, the harmony may need to change just to avoid sterility, even though a single chord would fit with all the notes. When the harmony must change, it should do so on the first beat of a bar, or possibly on the beat that divides the bar in half. So there is a complete interplay of melody, rhythm, and harmony that keeps things moving in a "sensible" way.

With these rules in mind, we can now divide our phrases into measures. Having decided that a phrase will contain eight attacks, let's subdivide the phrase into four groups. It hardly seems worth the effort to use the **BREAK** algorithm here (to see the entire family of four-number seeds whose entries add up to eight). In the same spirit used to select the other values in this example, I will choose 1 2 3 for the first three groups, and so be forced to use 1 2 3 2 for the complete specification ($1+2+3+2=8$). This says that the first bar of a phrase will have a single melody note, the next bar will contain the next two notes, the third bar will use the following three notes, and the fourth will contain two more. The next eight notes of the melody will be similarly distributed in the next four-bar phrase, and eight more in the phrase following that.

With the first three phrases accounting for 24 notes, only 12 (or 13 in the corrected version) remain to fill the last two phrases. Rather than divide these evenly, let's use a "deceleration" technique that keeps eight pitches for the fourth phrase and leaves only four for the last. In order to have the notes in the fourth phrase "slow down," the bar groupings will use the original values (1 2 3 2) but be sorted into decreasing order, 3 2 2 1. This will be followed in the last phrase with the decelerating pattern for four tones, 2 1 1 0, where zero implies holding over the last note to fill the last bar of this phrase. To humor my harmonization program, these values will need to be stated as 2 1 .5 .5—but we're getting too far ahead with such details, so let me just show some results in an attempt to prove that our melodies are indeed "legitimate." The complete methodology will unfold as we continue.

In transcribing the computer output shown in Fig. 9-10 to musical form Fig. 9-11, I chose a waltz (3/4) meter and assigned durations to the note groups in a purely subjective manner. Alternative harmonizations for the last note readily suggested the first and second endings. Figure 9-12 shows the results when the "corrected" melody was used with almost the same rhythm. In this case, the note grouping for the last phrase was changed to 2 1 1 1 to account for the additional note. That added note shifted all the notes following it with respect to the bar lines and was thus responsible for the harmonic differences in the last three phrases.

Of the melodies produced in the terminal session using the generalized algorithm (Fig. 9-9), the first four each had 55 notes. Similar treatment produced a different rhythmic setting than that derived for the 36 or 37 notes in the original, but one which could be used for all four cases. The notes are distinctly different in each of the four melodies, but identity of rhythm and the fixed nature of certain features of the algorithm, as explained above, make it impossible to tell which is the "theme" and which are the "variations." Here is the fourth one

```
8 DVLP2 ¯2 ¯7
```

which is shown musically in Fig. 9-13.

The first melody produced in that terminal session, using

```
1 DVLP2 2 3
```

has a high degree of "inherent tonality." Playing just the melody causes me to "hear" the prevalent harmony suggested by the chord symbols shown written above the melody in Fig. 9-14. As I increase my concentration, "inner voices" begin to stand out against the melody, suggesting the four-part treatment shown in the figure. (Note: The chord symbols apply only to the melody, *not* to the four-part voicing!)

But I can also ignore my instincts, specify some less usual harmonic structures and selection rules, construct a chordal rhythm that keeps things in motion, and let the computer turn out the not-so-obvious harmonic variation in Fig. 9-15.

Clearly pleased with the previous result and refusing to quit while ahead, I subjected the second melody, resulting from this expression:

```
       M
0 1 0 0 1 0 1 2 3 2 2 3 2 3 4 4 4 7 5 1 2 4 ¯1 1 2 4 ¯1 1 2 4
¯1 1 2 4 5 ¯2

       ATK
1 2 3 2 1 2 3 2 1 2 3 2 3 2 2 1 2 1 .5 .5

ORIGINAL TARGET M:
   0
   1 0
   0 1 0
   1 2
   3
   2 2
   3 2 3
   4 4
   4
   7 5
   1 2 4
   ¯1 1
   2 4 ¯1
   1 2
   4 ¯1
   1
   2 4
   5
   ¯2
   ¯2

M:   0:   C
H:        G↑    MA7

M:   0:   C↑    C
H:        C     LG7

M:   0:   C     C↑    C
H:        A↑    MI7

M:   0:   C↑    D
H:        G     LG7

M:   0:   D↑
H:        B     MA7

M:   0:   D     D
H:        D↑    MA7

M:   0:   D↑    D     D↑
H:        G↑    MA7
```

```
M:  0:  E     E
H:      F↑    LG7

M:  0:  E
H:      F     MA7

M:  0:  G    F
H:      F     MI

M:  0:  C↑    D     E
H:      E     MI7

M:  ¯1:  B   +1C↑
H:      A     LG7

M:  0:  D     E    ¯1B
H:      D     MI7

M:  0:  C↑    D
H:      G     LG7

M:  0:  E   ¯1B
H:      C     MA7

M:  0:  C↑
H:      F↑    LG7

M:  0:  D     E
H:      B     MI7

M:  0:  F
H:      E     LG7

M:  ¯1:  A↑
H:      A↑    MI7

M:  ¯1:  A↑
H:      F↑    LG7
```

Fig. 9-10. Output of a session in which the melody of Fig. 9-6 and 9-7 was harmonized.

```
2 DVLP2 ¯3 4
```

to the same treatment. Figure 9-16 is the computer's harmonization based on the same structures, selection rules, and chordal rhythm as in the above example.

The fifth melody produced in that session needed still different rhythmic groupings to accommodate 73 notes, but the same general method produced Fig. 9-17.

In all the automated examples, additional logic was needed to "voice" the harmony—that is, to decide which note in the chord should be lowest, next lowest, and so on. The next step would be to break up the chords or add to them in some way to derive a more musical accompaniment. But let
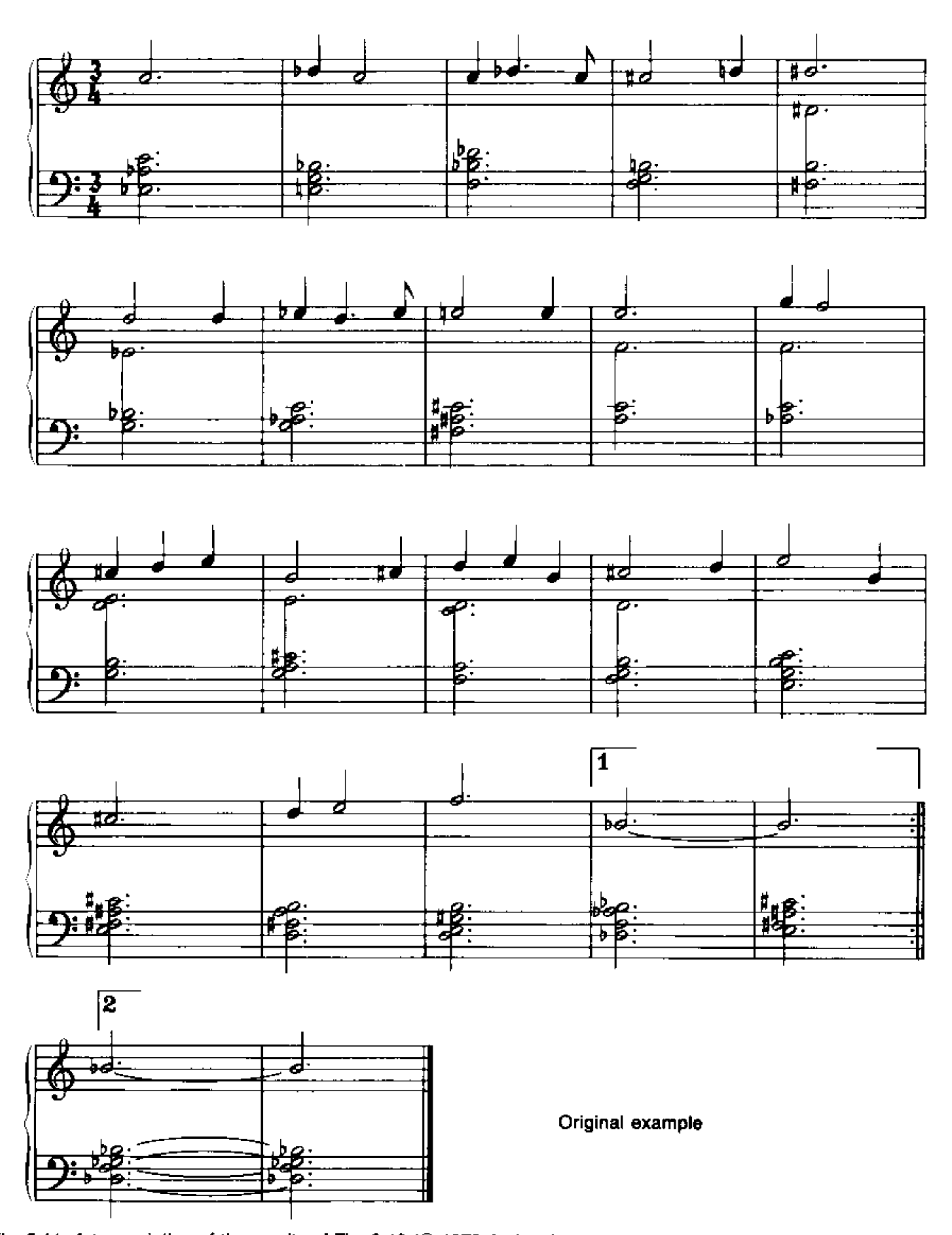


Fig. 9-11. A transcription of the results of Fig. 9-10 (© 1978 Jaxitron).
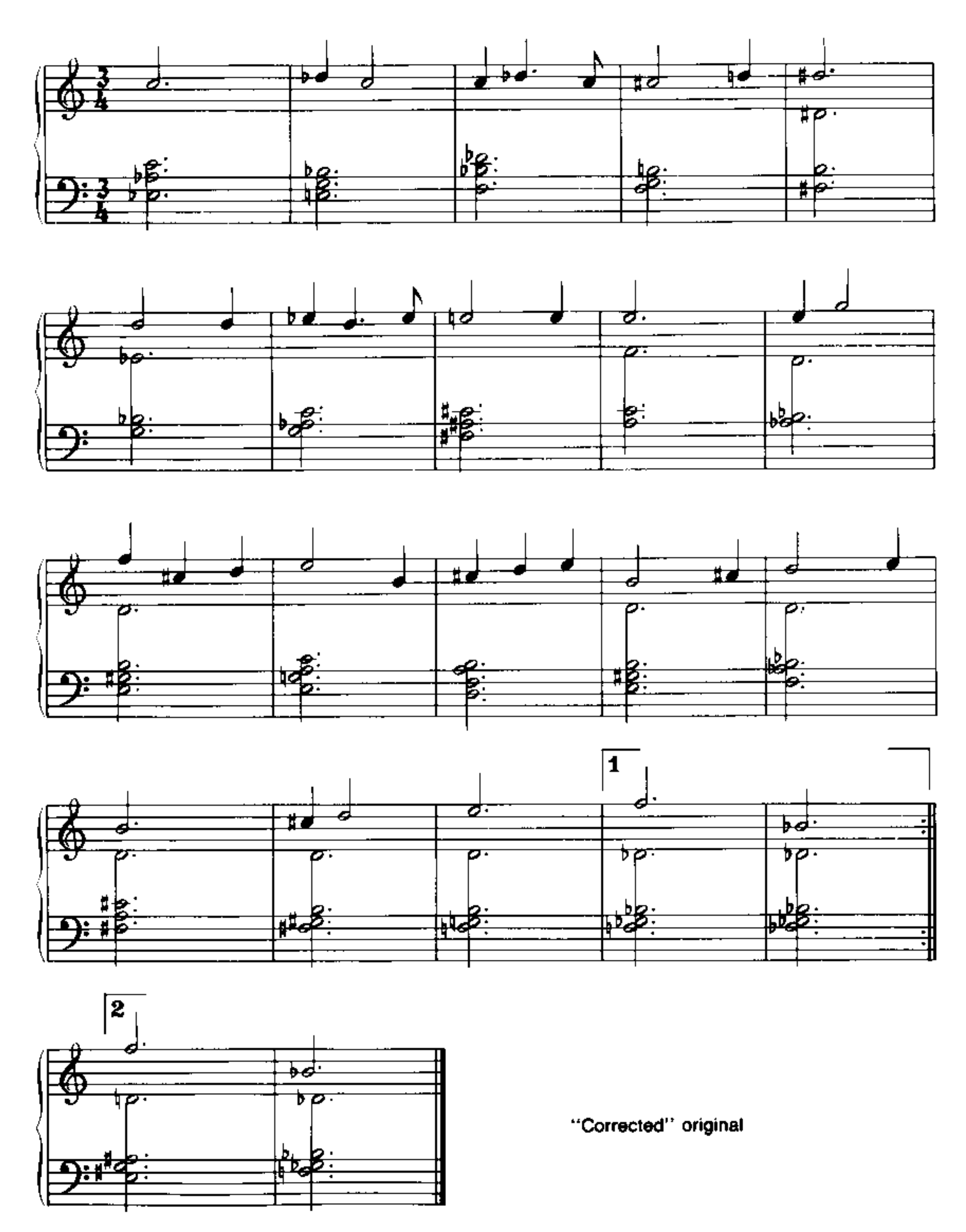


Fig. 9-12. Results of a harmonization session using the "corrected" melody (© 1978 Jaxitron).
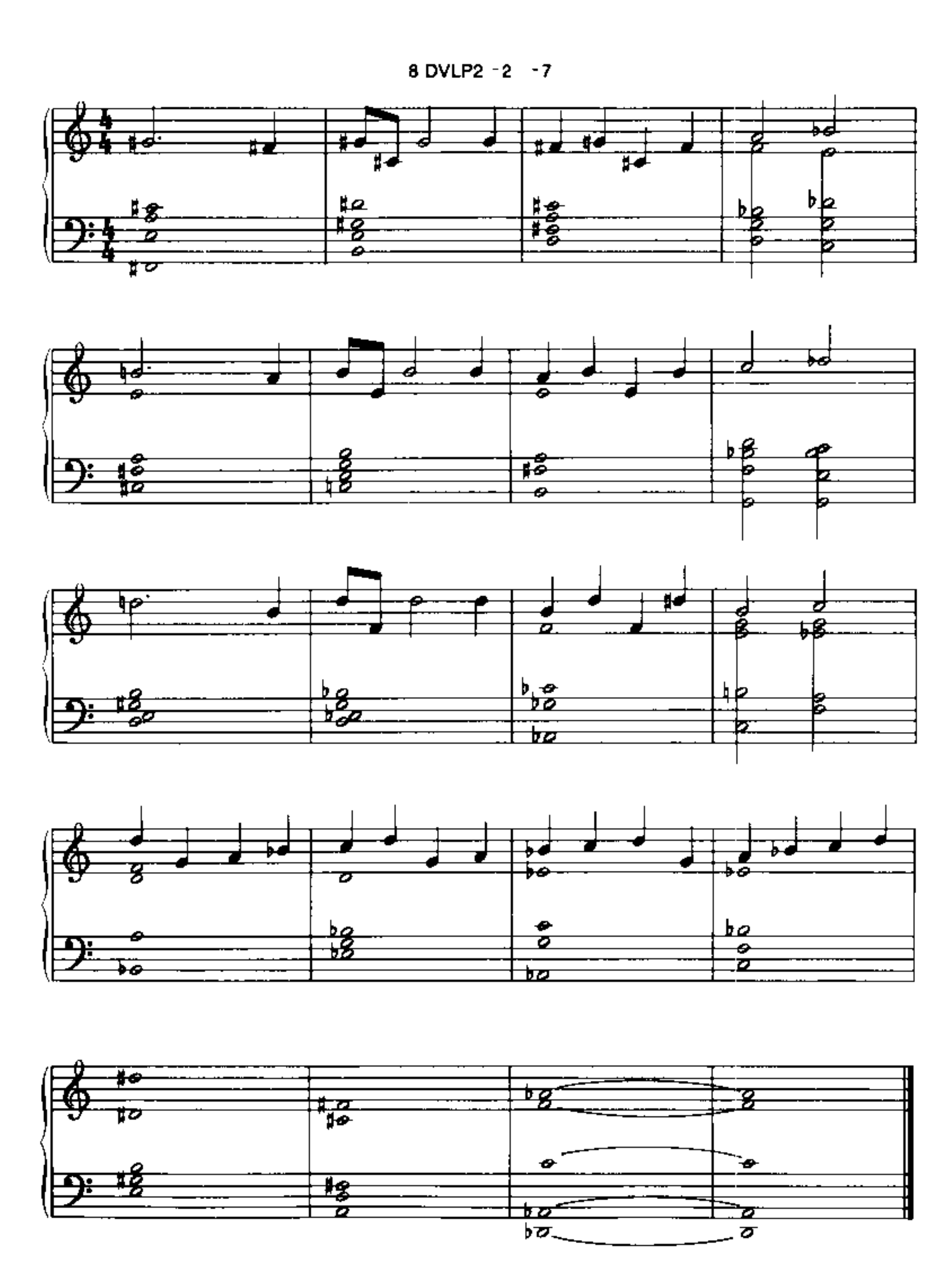


Fig. 9-13. Transcription of the result for 8 DVLP2 ⁻2 ⁻7 (© 1978 Jaxitron).
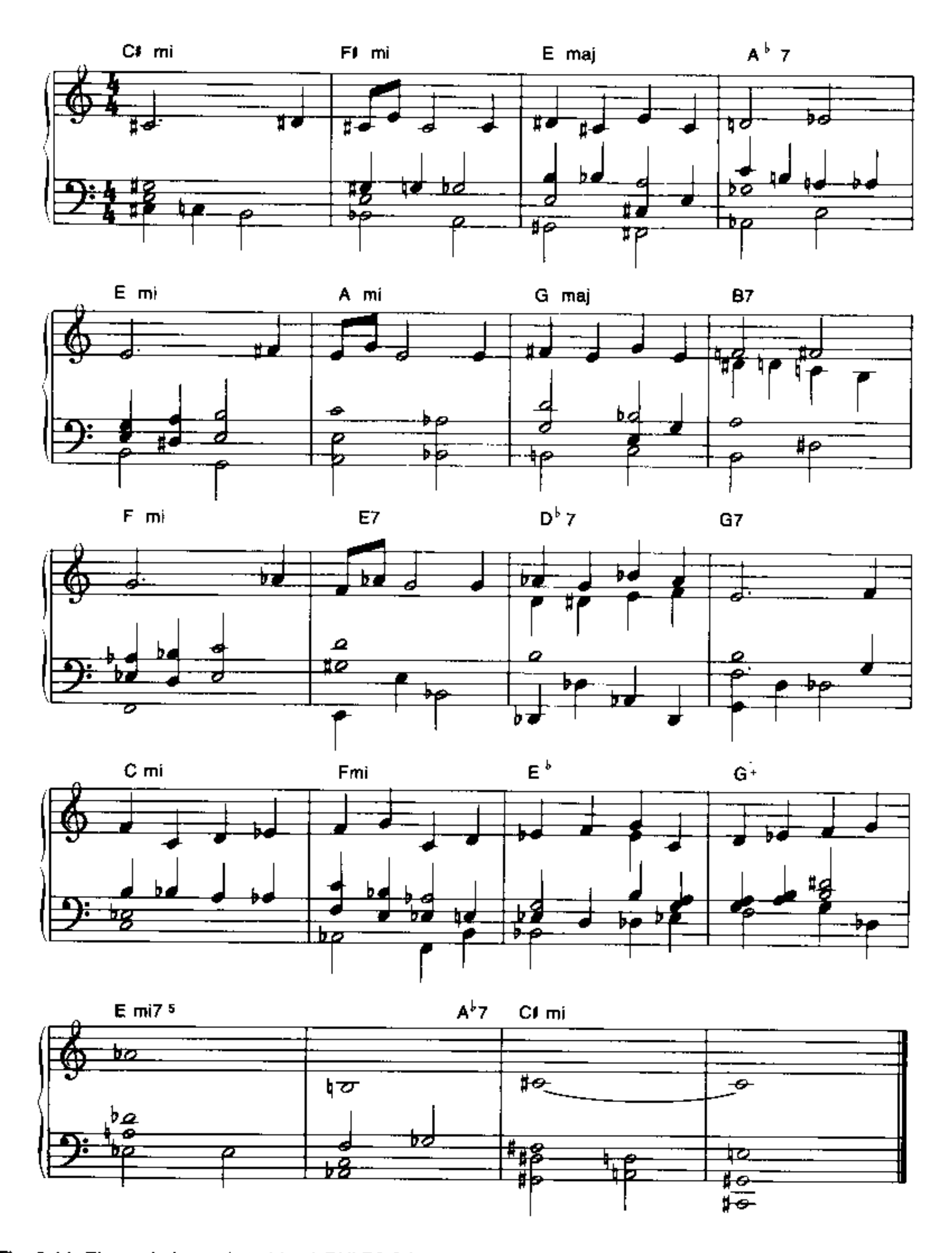


Fig. 9-14. The melody produced by 1 DVLP2 3 harmonized "instinctively" (©1978 Jaxitron).
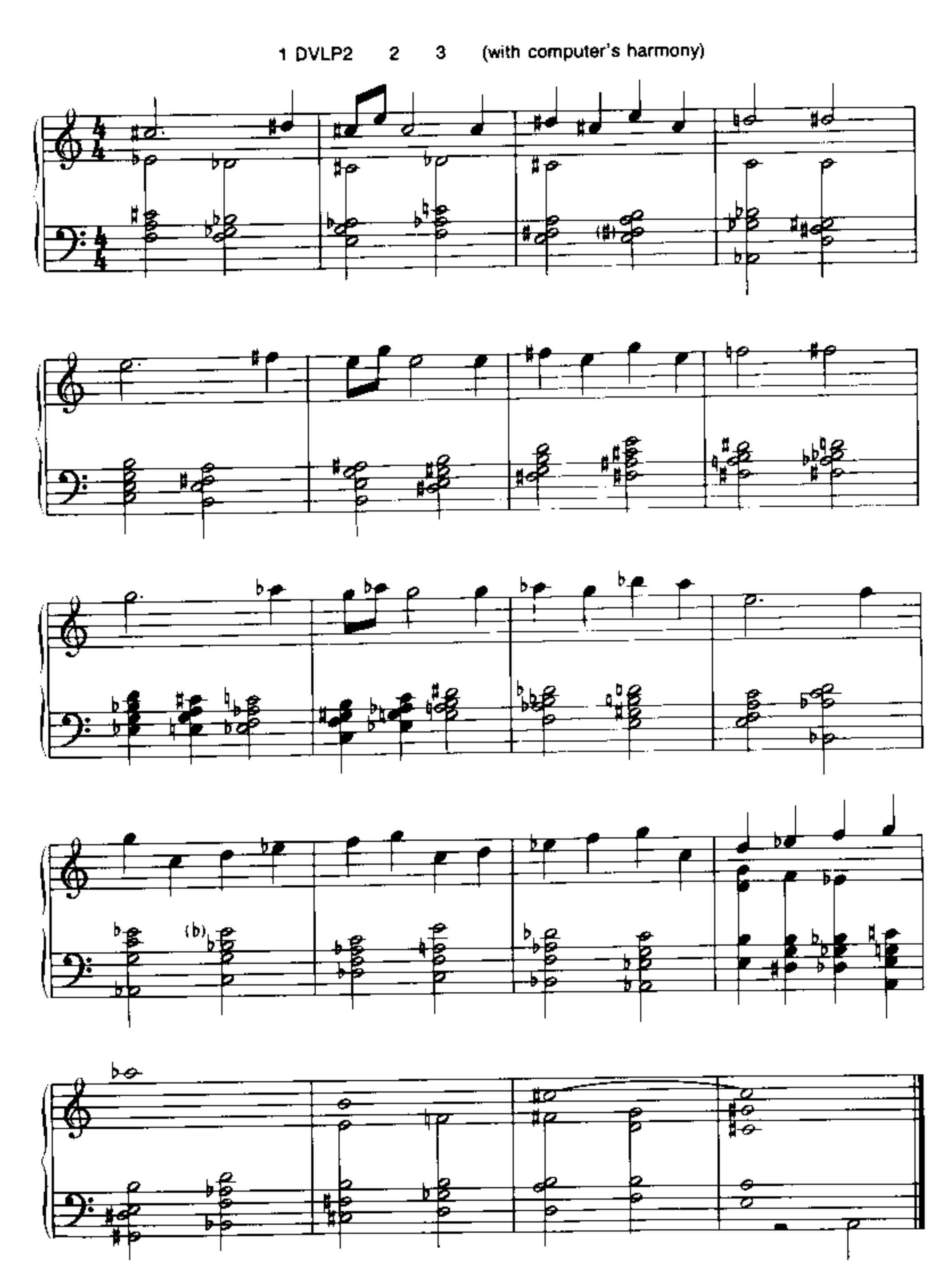


Fig. 9-15. The same melody as Fig. 9-14, harmonized using the computer algorithm (©1978 Jaxitron).

2 DVLP2 ¯3 4
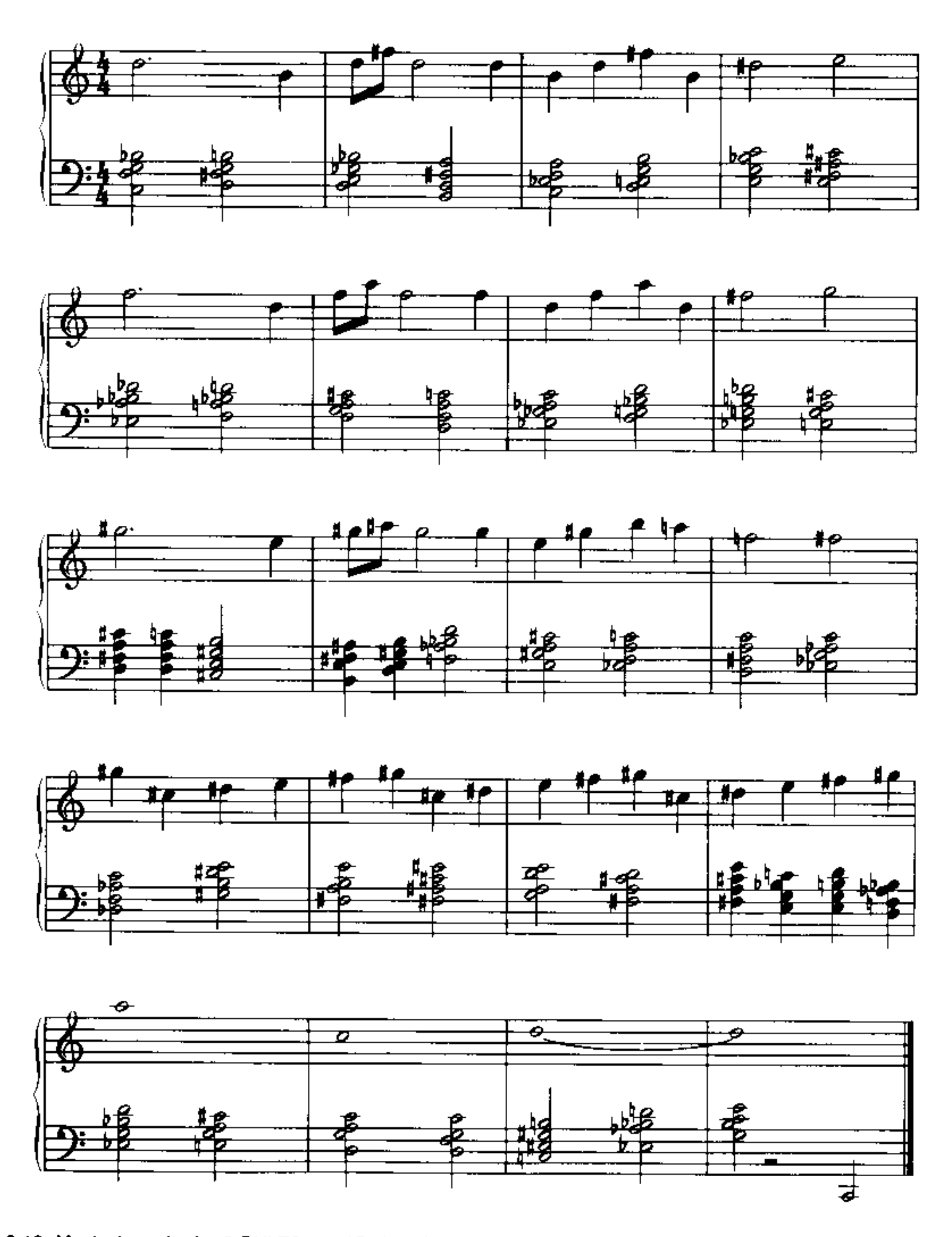


Fig. 9-16. Musical results for 2 DVLP2 ¯ 4 (©1978 Jaxitron).

5 DVLP2 2 2 1
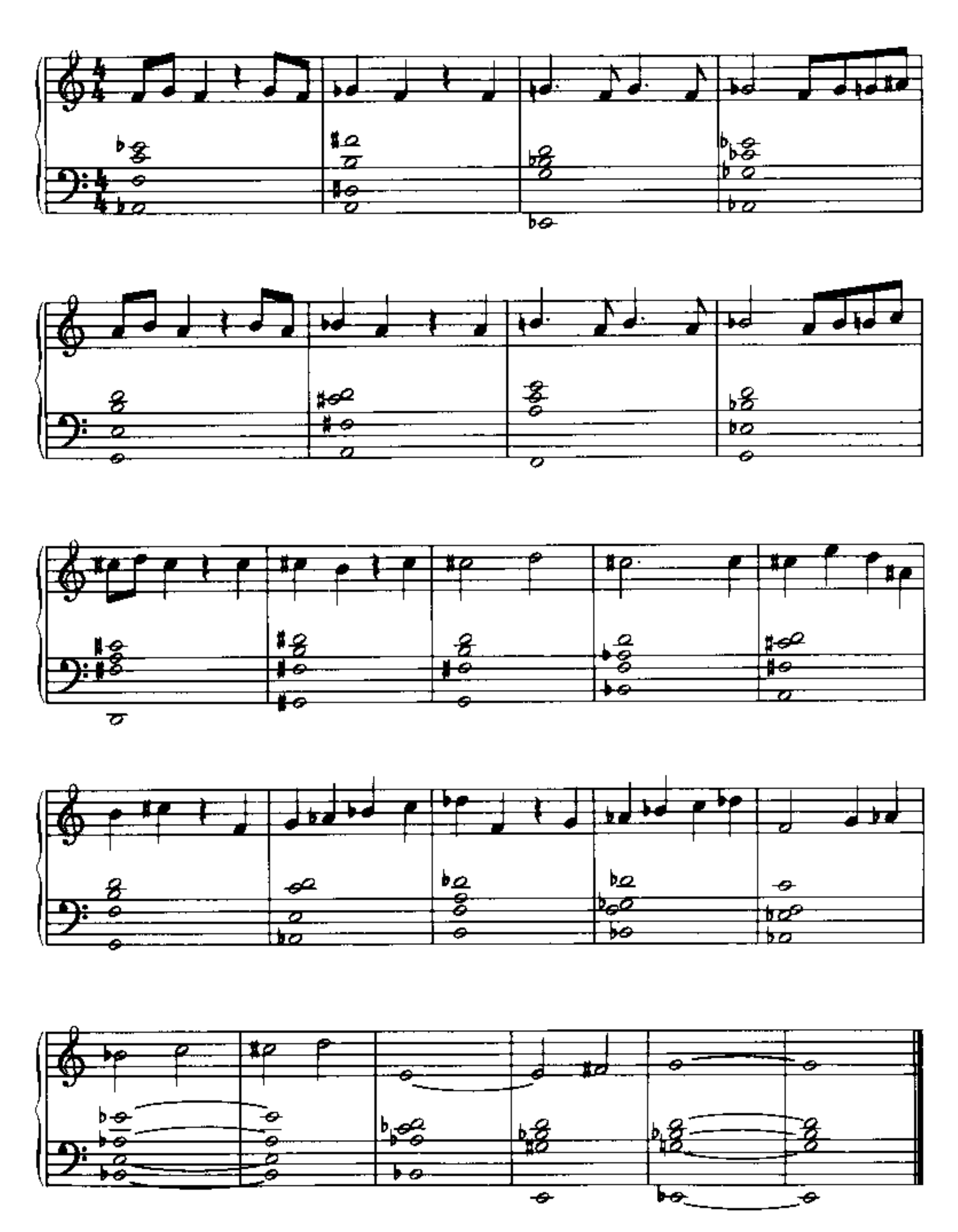


Fig. 9-17. Results for 5 DVLP2 2 2 1 (©1978 Jaxitron).

me reserve the one result not yet shown from Fig. 9-9 for a more complete demonstration in a later chapter.

It is too easy to be carried away here by introducing rhythmic and harmonic methods. As indicated earlier, our culture rarely indulges in melody for its own sake, and so, to demonstrate the "sensibility" in our results, I felt the need to display them in more complete form. It should be apparent now that there is more to a melodic line than building blocks and the mechanical fundamentals of "bricklaying," so we will now take a more musical approach to creating melody. We'll return to harmonic considerations after that.

10 Form

Pattern, structure, form, or whatever you care to call it, is by no means a novel idea in music. Form, or with more formality, temporal form, holds a prominent position in the study of composition. It is not a simple topic, and its study is usually based on analysis of exemplary works rather than on principles which encourage innovation.

In our chosen idiom, the problems of form often vanish when a composition is to fit a given set of lyrics. The lyricist's language suggests—in many cases, determines—the temporal structure of the composition. The first few words may provide a "natural" microrhythm for a melodic motif. This, along with successive motifs, will create an intermediate-level rhythm for a phrase. Phrases then fit together into the macrorhythm of a stanza or chorus. The term *form* often is applied on the grand scale that describes the sequence of macrorhythms in the entire work, but we will apply it to the entire structure at all levels of rhythm and pitch.

There is a traditional sense of rhythm at the intermediate level that strongly affects one's ability to devise original forms. This sense tends to force phrases to come in lengths of four, eight, or 16 bars. If you have so little experience with music that you don't feel this "natural tendency," I am torn between advising that you develop it or that you consider yourself fortunate and avoid it. Most lyricists do have this traditional sense, and may balk if a collaborator phrases their words in an "unnatural" way. At any rate, in what follows we will not be given any lyrics, but will construct a melody with temporal form as a primary consideration.

The melodic techniques demonstrated in the previous chapter satisfy certain *structural* requirements for sensibility. Many listeners will feel that the results by themselves—that is, without harmonization—do not meet their personal *harmonic* requirements for sensibility. Of course, they may not recognize this to be the source of their dissatisfaction, but melody, even when unaccompanied, must carry a strong suggestion of harmony for these people. Though melody is often referred

to as linear or one-dimensional by artists, its composition normally requires a multidimensional thought process. Composers somehow maintain simultaneous ideas on rhythm, harmony, and melody, and blend them into a particular form. Describing each of these subjects simultaneously derails my one-track mind. Still, I feel this is the place to introduce form, even though we have yet to begin a serious discussion of harmony. In the demonstration to follow, my tonal hypothesis concerning "scale-like" interval sequences will substitute for harmony in giving the results an apparent tonality. Later, when the resulting melody is harmonized, a different tonality will probably be observed.

A PROCEDURAL ALGORITHM

Let me begin with a listing of a function named **FORM** (Fig. 10-1). It serves to keep the composer's mind from wandering too far by isolating and keeping account of the more gross problems associated with form. It does not solve those problems. And now, to keep my mind from wandering, I'll describe an actual terminal session using this function. A complete transcript of the session and its results follows at the end of the chapter, in Listing 10-1.

```
      ∇ FORM ALFA;SYM;K;I;ROW;Z
[1]     ALFA←(ALFA≠' ')/ALFA
[2]     SYM←UNIQ ALFA
[3]     K←ALFA∘.=SYM
[4]     K←K×+\K
[5]     K←(K≠' ')/K←⍕(,K≠0)/,K
[6]     SYM←ALFA,[0.5] K
[7]     ROW←0
[8]     L1:K,K,'ENTER ',SYM[ROW;],K,K←⎕TC[1]
[9]     L10:I←⍞
[10]    →((1↑I)=' ⎕↑↓)')/L2,L3,L4,L5,L6
[11]    ⍎I
[12]    →(L10,L1)[4=+/'FORM'=4↑I]
[13]    L2:→((1↑⍴SYM)>ROW←ROW+1)/L1
[14]    →L50
[15]    L3:→(1=⍴,I)/L31
[16]    '' SHOW 1↓I
[17]    →L10
[18]    L31:SYM[ROW;],':'
[19]    N2P⍎SYM[ROW;]
[20]    →L10
[21]    L4:ROW←ROW-⌊/ROW,⍎1↑1↓I,'1'
[22]    →L1
[23]    L5:ROW←⌊/(¯1+1↑⍴SYM),ROW+⍎1↑1↓I,'1'
[24]    →L1
[25]    L6:S∆FORM←1+⎕LC
[26]    S∆FORM←⍳0
[27]    →L1
[28]    L50:⍎'Z←',¯1↓,SYM,','
[29]    'NAME Z (=',(¯1↓,SYM,','),')? AND/OR ''→0'' '
[30]    →L10
      ∇
```

Fig. 10-1. Function FORM guides the structural development of a composition.

The **FORM** function is invoked with a right argument that portrays symbolically a structural sequence. My opening statement in the session:

```
FORM 'AABA'
```

declared that I intended to devise four successive sections which were to be related to each other structurally as are the letters AABA. (Incidentally, this is the most common form for a song.) As with actual letters, where differences in script, font, or case don't change the identify of the letter, the various A-sections here may contain certain structural differences and yet be clearly recognized as A-sections.

The program replied to my invocation most intelligently by requesting that I next describe the first A-section:

```
ENTER A1
```

At this point, a user could set A1, as if it were the name of a variable, to any structure, perhaps to one already in existence—e.g., a global melody M derived from **DVLP2** or a similar function. However, my response was less direct; it may even appear absent-minded:

```
FORM 'ABCABD'
```

With the function already operating, I was calling for another level of execution in which the A1 section of the total theme should itself have the form "abcabd." Although, without lowercase characters available, it was indeed absent-minded of me not to choose different letters here to avoid confusion in what follows. But the program couldn't be confused, and it plowed ahead into this sublevel of execution of the **FORM** function, asking me to enter the first A-section for this case:

```
ENTER A1
```

ESTABLISHING THE FIRST MOTIF

My approach at this point has become so "standardized" that I am tempted to incorporate it into the **FORM** algorithm. The first appearance of any subsection (such as A1, B1, C1, etc.) goads me to establish a motif based on a "tonal" scale. Further, appearance of the first "A" subsection (or better, the first "a" subsection, to differentiate it from the A-section of which it is a part) prods me to define a *tonal kernel* that can be used for all the other first subsections. This approach has almost become a ritual through adoption of certain short APL functions, in particular, one named **MOTIF** (Fig. 10-2) and the **SCLI** and **SCLIT** functions already introduced.

The **MOTIF** function requires three variables: an implicitly available scale (SCL), an explicit starting point S in that scale, and an interval sequence I to be applied within the scale. The statement

```
X←25 MOTIF 1 2 3
```

would place in X the pitch numbers corresponding to

```
SCL[25 26 28 31]
```

which is the starting point (25) in the scale followed by successive scale entries one, two, and three steps away from each other.

Returning to the demonstration, recall the computer has asked me to enter an "a1" subsection. As just described, my first step then was to set up a tonal kernel for use in all the various first subsections:

```
KER←2 1 2
```

and this was followed by specifying the a1 motif:

```
    ∇ Z←S MOTIF I
[1]   Z←SCL[+\S,I]
    ∇
```

Fig. 10-2. Function MOTIF creates a melodic passage based on intervals in a scale.

```
A1 ← (SCLI KER) MOTIF ¯3 3 2 ¯1.
```

This established the scale and the motif in a single statement. Remember, the **SCLI** function not only constructs a ten-octave scale, but also returns the position of middle C in that scale. So here, the position of middle C in the scale based on 2 1 2 is the left argument—the starting note—for **MOTIF**. The right argument describes the motion in this scale to be down three scale steps from the starting point, back up three steps, up two more scale steps, and then down one step. (In demonstrations of this sort, I often select the numbers one, two, and three, and some patterning scheme that makes it easy to see where the motif ends with respect to the starting pitch. If you add the numbers in the right argument, you'll find the last pitch must be one scale step above the first one.)

After each such statement, the computer just sits quietly awaiting the user's signal that he has finished entering information for the current subsection and wishes to go on. That signal consists of an empty or blank line, which is what I now entered to say that the definition of "a1" was complete. The computer woke up just long enough to tell me to

```
ENTER B1.
```

This new request was for the first "b" part of the first A-section of the total form.

THE FIRST "b" SUBSECTION

With the first appearance of another subsection, we need to consider where the melody is, where we want it to go, and how to get it there. The next statement,

```
B1 ← (SCLI 1 ⌽ KER) MOTIF ¯1 1 1 1 1
```

grew out of these three considerations as follows.

The first note of this motif is determined by the left argument of the **MOTIF** function, which is the expression (**SCLI 1⌽KER**). Remember, **SCLI** returns the position of middle C in the scale, so this motif, like the first one, will begin on that note. Also recall, the first motif ended one scale step above the starting pitch, so the transition between the a- and b-motifs here will not appear awkward to a singer or listener. The scale for the previous motif was based on the kernel (**KER**) 2 1 2. Rotation of this kernel (1⌽KER) to 1 2 2 used as the right argument for **SCLI** produces a new scale that is effectively a transposition of the previous one. (Notice the sequence of intervals

. . . ⋮ 2 1 2 ⋮ 2 1 2 ⋮ 2 1 2 ⋮ . . .

will also be found in the scale containing

. . . ⋮ 1 2 2 ⋮ 1 2 2 ⋮ 1 2 2 ⋮ . . .

Two scales having the same interval sequence are either the same scale or one is a transposition of the other.)

My reasons for the intervals determining the structure of this "b" subsection in the right argument of **MOTIF** are more subtle. First, the previous motif leaped about; for contrast, this one will move scalewise. Second, the left argument causes a change in the tonality (the transposition of the scale), and this scalewise movement will make that quite clear to the listener. The actual sequence of scalewise intervals causes the starting pitch to be repeated, as it was in the opening motif: We move one step down from it and then back up to it. This is followed by three steps that take us as far above it as the opening motif took us below it, i.e., the first motif opened with an interval of minus three, while this one closes three steps above the first pitch.

Before going any further, I must admit sadly that this description makes my work at the computer sound far more objective and logical than it really is. All of this apparent structure is based on analysis of my statements rather than on the original reasoning used during the session. It is impossible to remember what really was going on in my mind at the time, although my subjective habits and experience always cause me to be concerned with structural considerations and "endpoints."

Whether the observed structure is the originally planned structure or a by-product of the original really doesn't matter as long as it's there. But I do always keep account of the absolute or relative positions of the first and last notes (the end-points) of each motif. Consistently, motifs must fit together in a "seamless" way, and, sooner or later, some motif will have to lead back into a repeat of the first motif. If any transition between motifs would trouble a professional singer, then it certainly wouldn't supply much comfort to more ordinary ears. As long as I know where I am in a scale (without actually knowing the details of the scale itself!), I will be able when necessary to force sensible motions from one point of reference to another.

THE "c1" SUBSECTION

After the single statement defining "b1," I entered another blank line and the program responded by asking for the first "c" subsection:

```
ENTER C1.
```

Once again a "first" subsection and once again I would use the **MOTIF** function. Of those three basic considerations (where we are, where we are going, and how we get there), the most important for this subsection may be the second one. The section to follow this one will not be a "new" one; "a2" will mark the second appearance of the "a" theme. So we may need to decide where "a2" will begin before we can bring "c1" to a close. This means that the second and third consideration must interact more deliberately than before.

To get started, I entered

```
C1 ← 1 ↓ (SCL ι ¯1 ↑ B1) MOTIF 2 ¯1 ¯3.
```

The left argument of **MOTIF** here, **(SCL ι ¯1 ↑ B1)**, made the last pitch of the previous subsection the first pitch here. Notice, the current scale **(SCL)** was not changed here. The "one drop" **(1 ↓)** at the very beginning of the statement removes the first note from the result so that, when "b1" and "c1" are played sequentially, the last pitch of "b1" will not be repeated although it serves as the starting point for the new interval sequence.

For the right argument, I wanted an interval sequence that would suggest a logical connection with what had gone before. So, a cyclic permutation of the "a1" intervals (¯3 3 2 ¯1) was used (2 ¯1 ¯3 3), but I wanted to repeat these "important" intervals, and so, to avoid a "built-in" repetition of a single pitch within the overall repetition, the last interval was dropped. This created only three notes in this section, but they would be stressed by repeating the whole pattern.

To increase the effect of this repeated, downward-moving, three-note motif and simultaneously imply a connection with the "b1" scalewise pattern, the last three notes of "b1" could be added in reverse order, giving three sequences of three descending tones. These particular notes extracted from "b1" offer another advantage. Notice that the three intervals opening the "c1" sequence total negative two (2 + ¯1 + ¯3). Because we began this section three scale steps above the opening pitch, the repeated three-note motif in "c1" (forming the first two three-note sequences) ends one scale step above that starting tone. Now look back at the interval sequence, "b1," which began on the same starting pitch. Its last three notes, when played in reverse order, will also end one scale step above the start. The pattern of three successive descending pitches ending on the same note as the two previous sets of three descending pitches has to convey a suggestion of "intelligence." It certainly gives that last note the aura of an intended target:

```
C1 ← C1, C1, ⌽ ¯3 ↑ B1
```

The apparent importance of the last pitch and its proximity to the first pitch of the "a1" motif suggested it would make an interesting starting point for the "a2" motif—repeating a motif transposed up one or two semitones gives it a "lift" psychologically if not literally. So this appeared to be a reasonable place to end the definition of the "c1" subsequence. But before going on, I wanted a little more information on the material already com-

posed in anticipation of developing the "second" subsections, "a2" and "b2." Single intervals are extremely important and useful as connectors in development. So instead of entering a blank line here, I asked to see the (chromatic) interval sequence actually used in "a1":

```
INT A1
```

(See Fig. 7-7 for this "one-liner.")

The reply, ¯5 5 3 ¯1, showed the equivalent in semitones of the opening sequence, ¯3 3 2 ¯1, in *scale* intervals. If I wanted to select interval from this set as being the most "significant," the value five, without regard for direction (up or down, plus or minus), strikes me as the one to choose. So, I decided to set this aside for further development by writing

```
NTRVL1 ← 5
```

In the same manner,

```
INT B1
```

showed me the "b1" motif contains the chromatic intervals

```
¯2 2 1 2 2
```

and the value two seems to come as close to capturing its "essence" as a single interval can. So I wrote

```
NTRVL2 ← 2
```

If forced to explain rationally how I can single out these intervals for special consideration, two features offer themselves. At least half the intervals in each motif are accounted for by that motif's representative interval, and each motif begins with the same pattern, moving downward through its interval and then upward through the same interval.

DEVELOPING THE "a2" AND "b2" SUBSECTIONS

Now I was ready to continue:

blank line
```
ENTER A2
```

As planned above, I wanted "a2" to begin with the last note of "c1." However, because it was already so well introduced, I saw no point in repeating this pitch at the start of "a2," so another "one drop" would be used to eliminate it. In addition, I decided to "develop" the "a"-motif here by appending to if the two opening intervals—the leap down and back:

```
A2 ← 1 ↓ + \(¯1 ↑C), (INT A1), 2 ↑ INT A1.
```

Having in mind only the interval form of the two "a" subsections, I wanted to see their pitches. Typing

```
A1
```

brought the response

```
0 ¯5 0 3 2
```

and

```
A2
```

was followed by

```
¯4 1 4 3 ¯2 3.
```

It was apparent from this that the last pitch of "c1," which is one *scale* step above the first note of the original "a" section, must be C♯ (one). Placing this value (one) in front of the numbers shown for "a2" reveals the five pitches of "a1" are all raised one semitone in "a2." We can also see the effect of the added down-up sequence on the end of "a2."

Next, after the ritual that began with my entering a blank line, I wanted "b2" to be simply a repetition of "b1" but transposed to follow "a2" the way "b1" followed "a1." If I were as analytic during composition as I am while "explaining," I could have made use of the **BGN** function to perform the transposition up one semitone. That not

```
      ∇ Z←M XPOS P
[1]     →(P[0]<0)↑L1
[2]     Z←(¯1↑P)+M−M[1↑P]
[3]     →0
[4]    L1:Z←M+(¯1↑P)−M[P[0]+⍴M]
      ∇
```

Fig. 10-3. Function XPOS transposes melodic segment M so that the entry in M indexed by the first entry in P will move to the pitch given as the second (last) entry in P.

being the case, Fig. 10-3 is the transposition function I did use.

Here, the left argument must contain the melodic vector to be transposed. The right argument is a two-element vector containing first a positive or negative index pointing to an entry in the melodic vector, and second, the pitch value that entry should have in the transposed version. So, by writing

```
B2 ← B1 XPOS 0, ¯1↑C1
```

I was saying that "b2" should be the same as "b1," but transposed so that its first pitch (index equals zero) is the same as the last pitch in "c1" (which was also the implicit starting point for "a2").

THE "d1" SUBSECTION

After another blank line, the computer asked me to enter the last character in the "A" plot:

```
ENTER D1
```

With the A-section divided into the form "abcabd," the a- and b-subsections can be seen to have principal roles. The c-subsection plays an important subsidiary part, providing development and also leading into a restatement of the a-motif. The current subsection (d1) is even more important, in fact, critical. In the overall AABA form, "d" ends each of the A-sections. This means it must lead smoothly from A1 to A2. Possibly with some modification, it must also connect A2 to the B-section. And finally, it must form (or else lead into) a satisfactory ending for the entire piece.

In "normal" composition, the composer treats such endings and transitions through techniques that depend on his seeing or hearing the notes he works with. This implies that for each possible combination of notes, he may have a specific approach. If at all possible, I would rather find an algorithmic method that leads to satisfactory results without the need to analyze and store an encyclopedia of situations and their treatment. One promising technique relies on the creation of a passage that has what I refer to as "neutral harmony." Such harmony seems to lead somewhere (or require resolution), yet that somewhere is ambiguous. This is why I call it neutral; the listener can't seem to pin down an exact tonality for the pitch sequence, although the general feeling of tonality persists throughout the passage. We can play such pranks on the observer by selecting just a few widely spaced intervals, based on the tonal kernel, to fill the octave in a "neutral" way—i.e., without accidentally constructing one of the more common harmonic structures (which we'll take up in later chapters).

For the kernel 1 2 1 2, the associated scale contains derivative intervals of three (2 + 1) as well as five (2 + 1 + 2). Through the use of the sum-scan operation, **+\KER** will serve as a dilated kernel to produce a sparse scale, related to the original one but containing only the intervals 2 3 5, or four pitches separated successfully by a major second, a minor third, and a perfect fourth. Any permutation of the expanded kernel could also be used, such as:

```
1 ⌽ +\KER or ¯1 ⌽ +\KER.
```

Instead of using **SCLI**, the function **SCLIT**, with a left argument of 12, would cause the total scale to repeat the same four tones in all octaves because the sum over these intervals is 10 and, no matter what their order, the next interval will reach a tone outside the reference octave from zero to eleven.

It could have been expected that I would turn this discussion in the direction of kernels and scales because "d1" is a "first" (and only) subsection and so practically commands me to use the function

MOTIF. So far, my thoughts have suggested something like

```
D1 ← (12 SCLIT ¯1 ⌽ + \ KER) MOTIF . . .
```

and now we must consider how to define the melodic line in the right argument based on the "neutral" scale.

The geometry of the a- and b-subsections shows that the last pitch (the one leading into this "d1" section) is already some distance above the lowest tone used in the melody, so the general motion of this motif had better be downward. (During this session, I carelessly estimated the distance to be one octave. Had I taken the time, I'd see it to be 11 semitones.) To avoid passing too quickly over our sparsely populated "scale" and to give the neutral harmony time to establish itself, the motion should be that of a descending "wave." Three executions of "up one, down two" (6 ⍴ 1 ¯2) should fill those requirements. As an afterthought, prefixing a negative one to this pattern will assure that we go no higher than the starting note for this motif. It will also carry over to this section a structural feature of both the a- and b-sections, viz., beginning the motif with a step down and an equal step back. So the call to MOTIF became

```
D1 ← (12 SCLIT ¯1 ⌽ + \ KER) MOTIF ¯1,
6⍴1 ¯2.
```

This statement would cause the pattern to begin on middle C. So, I next transposed "d1" to begin on the previous pitch, the last note of "b2":

```
D1 ← D1 XPOS 0,¯1↑B2
```

By adding the intervals here (¯1 + 1 + ¯2 + 1 + ¯2 + 1 + ¯2), we see the total motion goes through four scale steps, which would take us down exactly one octave and (according to my bad estimate) would duplicate the lowest note already used.

All that "d1" now needs is an ending motif that will be flexible enough to allow the connections and endings described above. It should be based on "important" microstructures of the melody in order to appear as an integral part of it rather than as an appendage. Thus, I decided the ending micromotif would consist of the two special intervals. Its first appearance would be right at the end of "d1":

```
E ← + \ (¯1 ↑ D1), NTRVL1, NTRVL2.
```

This form makes the last pitch of "d1" its starting point. The three pitches can be repeated with transposition to lead wherever necessary. But just in case a longer connector might be needed, I decided to set aside the last three pitches of "d1" that lead into the first pitch of the ending, E:

```
LST3 ← ¯1↓¯4↑D1.
```

Immediately following this, the ending was attached without repeating the connecting note:

```
D1 ← D1, 1↓E
```

COMPLETING THE A-SECTION AND MAKING ENDS MEET

When the next empty line was entered, the computer responded in cryptic fashion with

```
NAME Z (=A1, B1, C1, A2, B2, D1) ?
AND/OR '→0'
```

This was my way (as programmer) of reminding myself (as user) that the program had completed its assigned task. It had created something called "Z" which now contained the subsections shown in parentheses. The question mark was asking me to decide whether I wished to save this result (by renaming the local variable Z) before telling the program that I was finished (by typing the branch-to-zero statement). To comply, the result was given a suitable name for a "first A" section:

```
FRSTA ← Z
```

and then I said, "That's all," by entering →0.

Having completed executing **FORM 'ABCABD'**, the computer now found itself back executing the first phase of **FORM 'AABA'**, still awaiting specification of the first A-section. So once again it asked me to **ENTER A1**. The expected conflict between symbols had now occurred, so to make certain the program was not confused, I asked to see **A1**. The reply showed that "A1" still contained the first "a" subsection:

0 ⁻5 0 3 2.

Then, on requesting a display of the entire **FRSTA**, the whole numeric sequence just constructed in the subphase of execution appeared:

0 ⁻5 0 3 . . . ⁻6 ⁻1 1.

I now had to decide whether this **FRSTA** would serve as the complete A1 and lead smoothly into A2, which would be a repetition of the first A-section. That is, do the last few values of the sequence, which comprise the E motif, lead in a tonal way back to the beginning of the sequence? No . . . at least not for the kind of tonality used in the rest of the melody. For the last few pitches and the first few to appear to be part of one scale-like tonality, their intervals should not include two successive semitones. Notice the last two pitches and the very first fall into the ordered sequence, ⁻1 0 1, where each separation is indeed a semitone. (When we examine tension and its relation to harmony and tonality, we'll see a quantitative reason for rejecting such sequences.)

After staring at the numbers in the E-motif and the intervals that went into its formation, **NTRVL1** and **NTRVL2**, a relationship came to mind that caused me to enter

E ← E + NTRVL2.

If the original E were immediately followed by this transposed E, the interval between the connecting pitches should be **NTRVL1**. Further, the added pitches would meet the beginning notes of **FRSTA** in an "easy" tonality. To make sure I hadn't miscalculated, I asked to see E. The reply

⁻4 1 3

confirmed my "fingerithmetic," and I tacked this new motif onto the end by specifying

A1 ← FRSTA, E.

Now, typing just the quad symbol caused the program to display the name of the current section of the form under construction and the pitches contained in it:

```
□
A1:
0:  C ⁻1G  +1C  D♯ . . .
                     . . . ⁻1F♯ B  +1C♯ ⁻1G♯
                     +1C♯  D♯
```

The last six notes, consisting of the two successive appearances of the E microstructure, need to account for three particular concerns here—and they do so as follows.

First, the two E segments meet in a way that "preserves tonality" in that every sequence of three or four notes can be easily harmonized by common structures. Second, the problem of leading into A2, which would begin as a repetition of A1 with no transposition, appears to be solved. The connecting interval here from the last D♯ to the initial C is a minor third (three semitones) that can be derived from the generating kernel, and tonality is preserved over the last E segment and the beginning of the "a1" motif.

The third concern questions the flexibility of the ending motif. Can A2 have a slightly different ending so as to mimic A1 and yet provide an interesting "surprise" leading to the B-theme as well as to a final ending? As long as the B-section is undefined and no thought has been given to the final ending, that question is somewhat rhetorical. But it serves to guide the development from this point.

THE A2 SECTION

To make A2 a repetition of A1, I decided to

include as part of the repetition some of the logic used in A1. That is, A1 was defined as **FRSTA** catenated to E transposed through **NTRVL2**. So, those same steps were taken here. The expression

```
E ← E + NTRVL2
```

transposed the already transposed E through the same interval, i.e., up another whole tone. Then

```
A2 ← FRSTA, E
```

completed the task.

Another quad symbol was typed and the A2 section was displayed, showing the last six notes to be

```
. . .  ¯1F↑  B  +1C↑ ¯1A↑ +1D↑  F.
```

This shows that the original E (the first three notes here) connects to the doubly transposed E (the last three notes) by moving down a minor third (the same connecting interval that linked the singly transposed E to the beginning of A2) and tonality is still preserved.

Having invested time and energy in getting this far, I typed a right parenthesis, causing the program to interrupt itself and have the APL system program display the point of interruption in the function:

```
FORM[26].
```

This allowed me to enter the system command

```
)SAVE.
```

On completion of the command, the time, date, and name of the APL workspace were displayed. Then typing

```
→□LC
```

caused execution to continue from the point of interruption. To remind me of the exact position in the logical sequence, the program reprinted the message:

```
ENTER A2.
```

I had finished defining this section before saving the workspace, but had not indicated that was the case by entering an empty line. I did so now.

THE B-SECTION "BRIDGE"

After the trying session just completed, the idea of shaping this section from a single repeated motif had considerable appeal. So the form would be "aa," or better

```
FORM 'XX'
```

to avoid more confusion over symbols. Dutifully, the program replied with

```
ENTER X1
```

Once more a "first" subsection was to be defined, and once more I would use the **MOTIF** function. The tonality would again depend on the original kernel, but now repeated in each octave through the use of

```
12 SCLIT KER
```

as the left argument of **MOTIF**. The choice of intervals for the right argument will again appear more circumspect than it really was.

I decided that the x-subsection should contain about a dozen pitches (eleven intervals) followed by a connective phrase or ending. For a micromotif, I chose the sequence of scale steps: up three, down one, down one (3 ¯1 ¯1). Adding these intervals shows a net motion up one scale step. If we use this pattern three times, the first nine intervals will take us three steps above the starting pitch (not yet specified) and leave two intervals to be chosen for approaching some target pitch. I chose a target one

step below the first pitch and the intervals minus five and plus one for getting there. Thus, the next statement was

```
X1 → (12 SCLIT KER) MOTIF (9ρ ¯1 ¯1),
¯5 1.
```

To see the chromatic intervals comprising this motif, I typed

```
INT X1
```

The reply,

```
5 ¯2 ¯1 5 ¯2 ¯2 5 ¯1 ¯2 ¯9 2
```

showed a satisfying abundance of the "special" intervals five and two (**NTRVL1** and **NTRVL2**). Only the negative nine stands out uniquely as a possible source of discomfort, but, because that is the inversion of a minor third used as a "connector" earlier, I decided to accept it.

Recall, in designing the a-section, two contiguous micromotifs were set aside for possible further use, LST3 and E. Having been neglected until now, the first of these was chosen to connect the parts of the B-section. **E** would be made to lead into LST3 through the interval, **NTRVL1**, and **X1** would be transposed so as to lead to E through **NTRVL2**:

```
E ← E XPOS ¯1, NTRVL1 + 1↑LST3
X1 ← X1 XPOS ¯1, (1↑E)−NTRVL2
```

There remained the need to transpose this whole sequence so that it would follow A2 comfortably. I asked to see how far the last pitch of A2 was from the first note of X1:

```
INT (¯1↑A2), 1↑X1
```

I wish I could take credit for having planned the response, "¯3." Without transposing at all, the same interval that connected the A-sections would also connect A2 to B.

Before appending the connecting motifs to the end of X1, the current X1 needed to be preserved for designing X2:

```
XX ← X1
```

Now, X1 could be reset to include E and LST3:

```
X1 ← XX, E, LST3
```

With the "XX" form, it is always more interesting to have each X in a different key, and often the last pitch of the first X makes an excellent starting point for the second one. So, following the programmed **ENTER X2** prompt, such a beginning was specified:

```
X2 ← (XX, E) XPOS 0, ¯1↑X1.
```

The plan here was to construct X2 with the featured motif catenated to E, as before, but to adjust the LST3 ending through transposition to meet the following A3 in a fitting manner.

Expecting the last A-section to begin on middle C, just as the first two A-sections had, I asked to see the last pitch of the pattern just defined:

```
¯1↑X2
```

The answer was a brief "8," indicating the current setting of X2 ends eight semitones above middle C. That is only four semitones below the C one octave above the starting pitch. That, coupled with certain information about the E motif that was still fresh in my mind, caused me to aim for this higher C as the target pitch that would end the X-motif and to forget all about using LST3. The last A would then begin one octave below; while the skip of an octave can be troublesome for many performers, it seldom strains anyone's musical intellect.

I remembered observing in the ending to the first A-section that the second pitch of the transposed E was the same as the last pitch of the original, preceding E. Instead of examining the listing to see what the transposition had been (plus

NTRVL2), I defined it "the hard way":

```
X2 ← X2, E XPOS 1, ¯1↑X2.
```

The right argument of XPOS here would transpose E so that its second note, E[1], would be the same as the last note of the current X2. If my reasoning was correct, catenating this to X2 would bring the last pitch a whole tone nearer the target pitch, so I asked to see

```
¯1↑X2
```

and was rewarded with the desired answer, "10." This meant that I could repeat the previous command:

```
X2 ← X2, E XPOS 1, ¯1↑X2
```

in order to reach the target C, one octave above middle C.

Entering an empty line brought the end-of-execution response:

```
NAME Z (=X1,X2)? AND/OR '→0'
```

to which I replied

```
BRDG ← Z
```

which named the result BRDG (for bridge) instead of B1.

After typing →0, the program asked me to ENTER B1. Anxious to end this prolonged session, I hurriedly entered another empty line, thinking that B1 had just been completed. As the message

```
ENTER A3
```

appeared, I realized my oversight. By entering just an "up arrow," I told the program to return to the definition of the previous section of the form. It responded by asking me once more to ENTER B1. This time I got it right; I set B1 equal to BRDG, and followed that with a quad symbol in order to see a display of the entire B-section and make sure my haste hadn't caused any other problems.

```
B1 ← BRDG
□
B1:
0:  D    G    F    E  . . .
      . . .  F    A↑  +1C.
```

Everything looked fine, especially that final pitch — +1C!

THE FINAL A-SECTION

The simplest way to end the piece would have been to set A3 equal to A1. But I was uneasy about doing so because the first two A-sections as well as the B-section all ended with the E-motif, and in each case this led to the next section. Through proper use of rhythm and harmony, a listener could be made to accept a final E-motif as a legitimate closing pattern. But it might be more interesting if that last E led into the initial "a1," and the rhythm and harmony molded that into a final ending. Because the four intervals that defined the opening motif ("a1") established the first five notes of A1, I typed

```
A3 ← A1, 5↑A1
```

Then a quad symbol initiated a listing of the last A-section. Following that, a blank line brought forth the prompt

```
NAME Z (=A1, A2, B1, A3)? AND/OR '→0'.
```

Instead of answering in the usual way, I asked, "How far must the entire sequence of notes (in all four sections of the melody) be transposed so that the complete range of pitch is centered on G above middle C (pitch number seven)?" Of course, I had to phrase the question a bit differently:

```
II ← (A1, A2, B1, A3) FITRNG 7.
```

(I'll show the **FITRNG** function in a moment.) I then asked to see the result:

```
II
```

and was curtly told:

```
4
```

So in order to capture the pitches in a range that would make their representation in treble clef easier on the eye than on my vocal chords, I requested that each section be printed transposed through the interval II:

```
N2P II + A1
 .
 .
N2P II + A2
 .
 .
etc.
```

The function **FITRNG** (Fig. 10-4) returns the interval needed to transpose a given melody (the left argument) so as to FIT its RaNGe to the condition specified in the right argument. That right argument contains a numeric pitch. If that is all it contains, or if the pitch is followed by a zero, the returned interval would "center" the melody on the given pitch, i.e., the transposition would make that pitch lie midway between the highest and lowest notes of the melody. If instead, the pitch number is followed by a negative one (¯1) or a positive one (1), the returned interval would transpose the melody so that its lowest or highest pitch, respectively, would be the given pitch.

```
    ∇ Z←M FITRNG N;MN;MX;A
[1]   N←2↑N
[2]   MN←⌊/M
[3]   MX←⌈/M
[4]   A←MN,(⌈.5×MN+MX),MX
[5]   Z←N[0]−A[1+N[1]]
    ∇
```

Fig. 10-4. Function FITRNG finds the interval needed to transpose melody M according to the conditions specified in the right argument.

THE FINISHING PROCESS

Rhythmization and harmonization, not necessarily in that order and not necessarily ordered at all, must follow. The first step taken here was to transcribe the melody into musical notation with tentative bar lines separating the various motifs (Fig. 10-5). I had to check back over the listing of the session to see how many attacks went into each section and, in cases where a starting pitch was dropped to avoid repetition, I had to decide which of the adjoining subsections would contain it. (Notice, all references to middle C in the description have been transposed up four semitones to the pitch E.)

While each of the segments has observable harmonic potential, there is nothing here that suggests to me any sort of meter. But now I must point out something that may seem ironic at first. From the normal perspective, the topics rhythm, melody, and harmony, in that order, run from most obvious to most obscure. In the most primitive civilizations, music consists of rhythm alone. As technology develops, instruments evolve that are capable of producing increasingly controlled melody. Our concept of harmony is unique to Western civilization. In the process of becoming a musician, one is introduced to these subjects in this same order. Yet when it comes to treating the subject cybernetically, trying to establish objective logic that mimics subjective methods and preferences, the order of the topics seems reversed. The more "civilized" the topic, the more rationally objective is its concept, and the easier it is to express its underlying logic.

As a result, when I describe my harmonic methods, I can be quite precise. In the previous descriptions of melody, it required more effort to make my subjective manipulations of pitches appear to be objectively methodical. In applying logic to rhythmic attacks, I seldom use methods so complex that they can't be carried out mentally. But

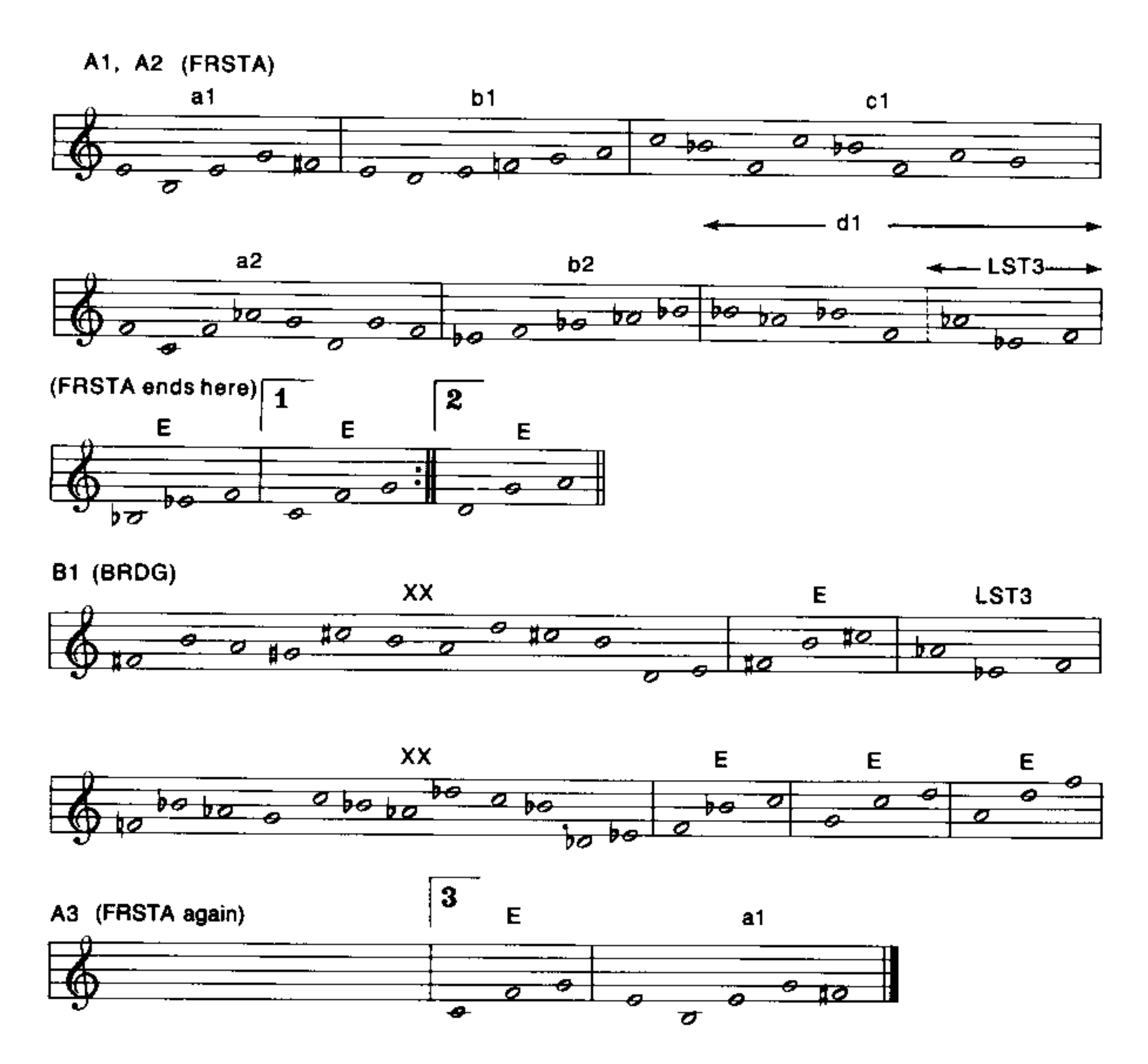


Fig. 10-5. The resulting melody showing the notation used in the terminal session (Listing 10-1) and the tentative placement of bar lines.

then, in assigning durations to the attacks, I can't even pretend to understand my "logic" well enough to program it or any model of it! Now with a clear conscience, I can describe how I attack the problem of meter.

With tongue firmly in cheek, I let the first motif suggest a tentative meter. The five attacks in the opening motif would suggest one measure of 5/8 time if not for our dedication to the MT idiom here. By stretching the last attack to fill a quarter note, a bar of 3/4 time seems respectable. I then try to apply this to the subsequent sections. As I proceed, sooner or later, I am forced to add additional bar lines and insert temporary changes in meter. This first effort ordinarily lets me get familiar enough with the material to work out a more serious construction. However, to show the spirit behind the tongue-in-cheek phase, here are my results placed
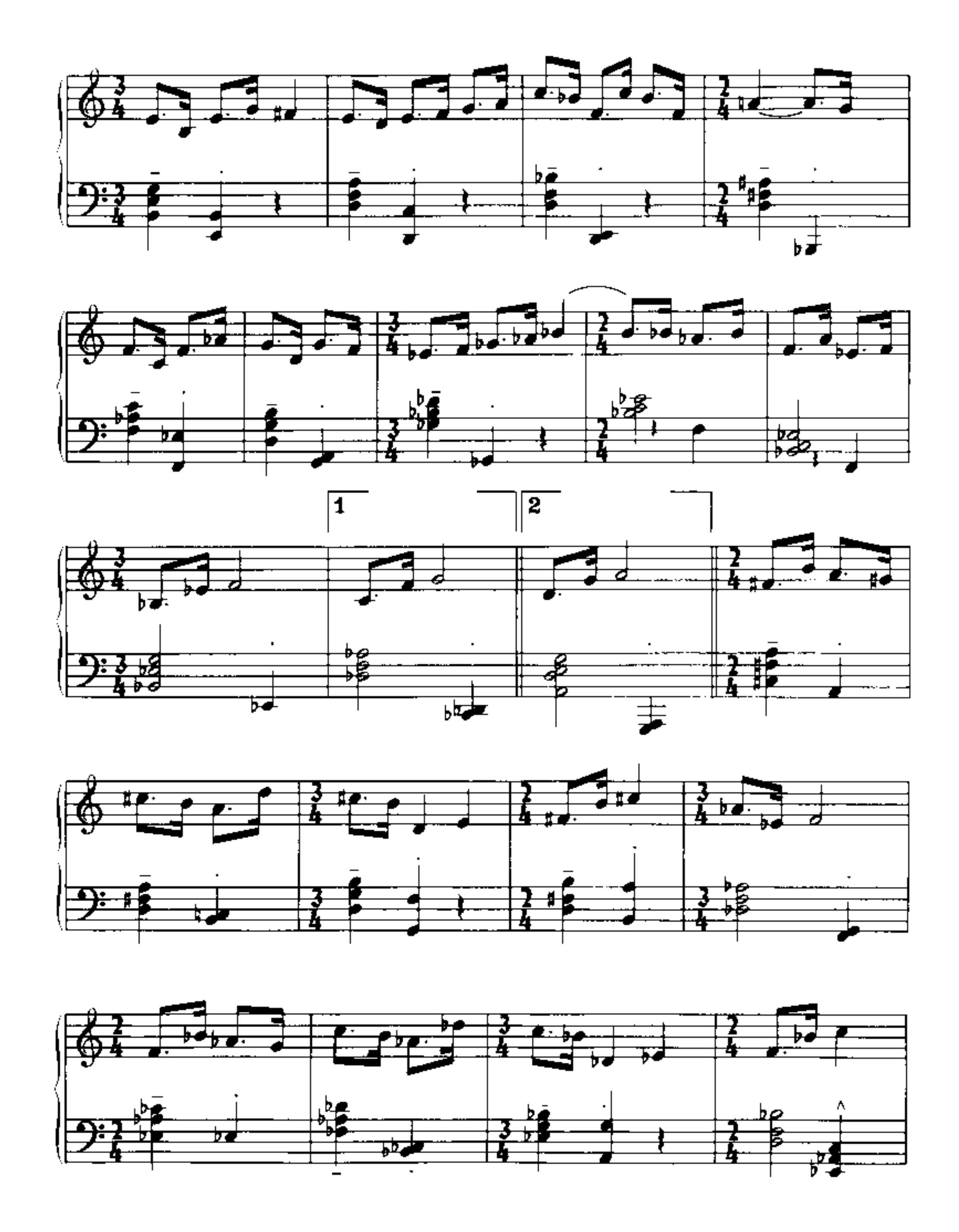


Fig. 10-6. A tongue-in-cheek development to explore the possibilities for meter and harmonic setting (© 1978 Jaxitron).
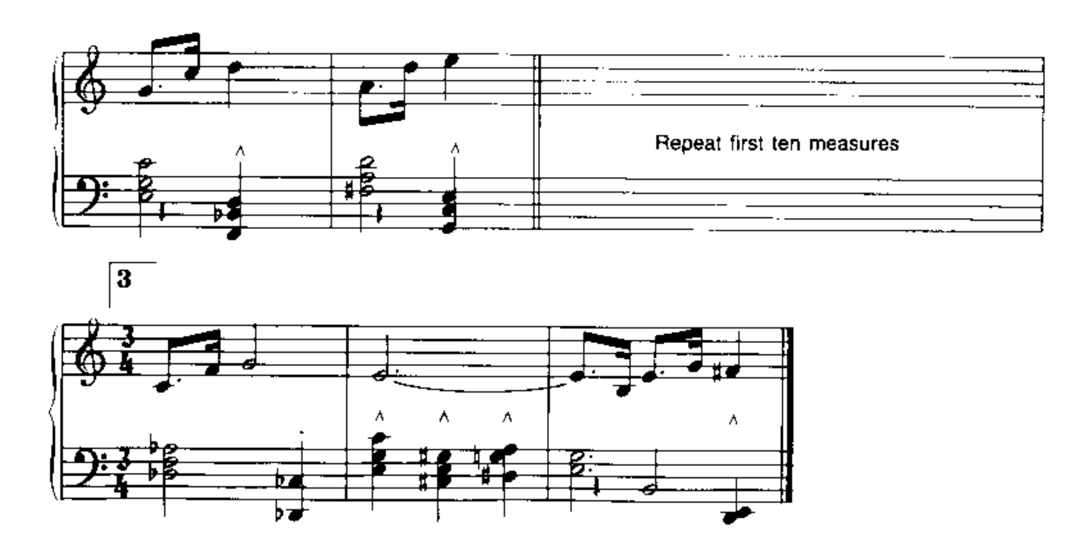


Fig. 10-6. Continued.

in a Shostakovitch-like setting (Fig. 10-6).

By the time I could "hear" this without looking at the music, I could "feel" how it could be made suitable for MT. Unfortunately, I see no way to describe this process other than to say that if the above piece were performed manually, by computer, or in a composer's mind, then the setting shown in Fig. 10-7 would appear as a "logical" development. The new harmonization was worked out with the computer after complete rhythmic definition.

Just before working this out, I had written the music whose rhythm is shown in the example in Fig. 7-30. Later, after hearing the above piece a few times, I recognized how the previous rhythm, not yet erased from my subconscious memory, had imposed itself here. I then rewrote the rhythm to paraphrase the older one even more closely; submitting that to my harmonization program produced the version shown in Fig. 10-8.

ADDITIONAL REMARKS

I am often asked why I don't automate the entire process of composition instead of depending so much on a "musically informed" user. I think the reader, having gone through the last two chapters, is now in a better position to appreciate the situation.

The functions **FORM** and **DVLP2** are poles apart: **FORM** gives the user tremendous freedom as well as tedium, while **DVLP2** relieves him of both. Of course, you can freely alter and experiment with both functions. In my own work, a function named **DVLP** is usually devised (just as was **DVLP2**, the 2 added to distinguish it from my current **DVLP** function), and then **FORM** is used to create the larger structure based on the results of **DVLP**. Much effort can go into the design of such functions because the act of composition requires that there must be, for every choice along the way, certain restrictions, a certain amount of liberty, and even some measure of license! Everyone will seek his or her own middle ground in which freedom and tedium must be traded off against each other.

Conventionally, composers select rhythms, strings of pitches, and devices of various sorts to develop a piece of music. There is nothing inherently more abstract about choosing strings of numbers and functions of various sorts to accomplish the same thing. In fact, such an approach introduces new perspectives, but the old problems of selecting material, developing fragments, and interconnecting them do not vanish. In the newer
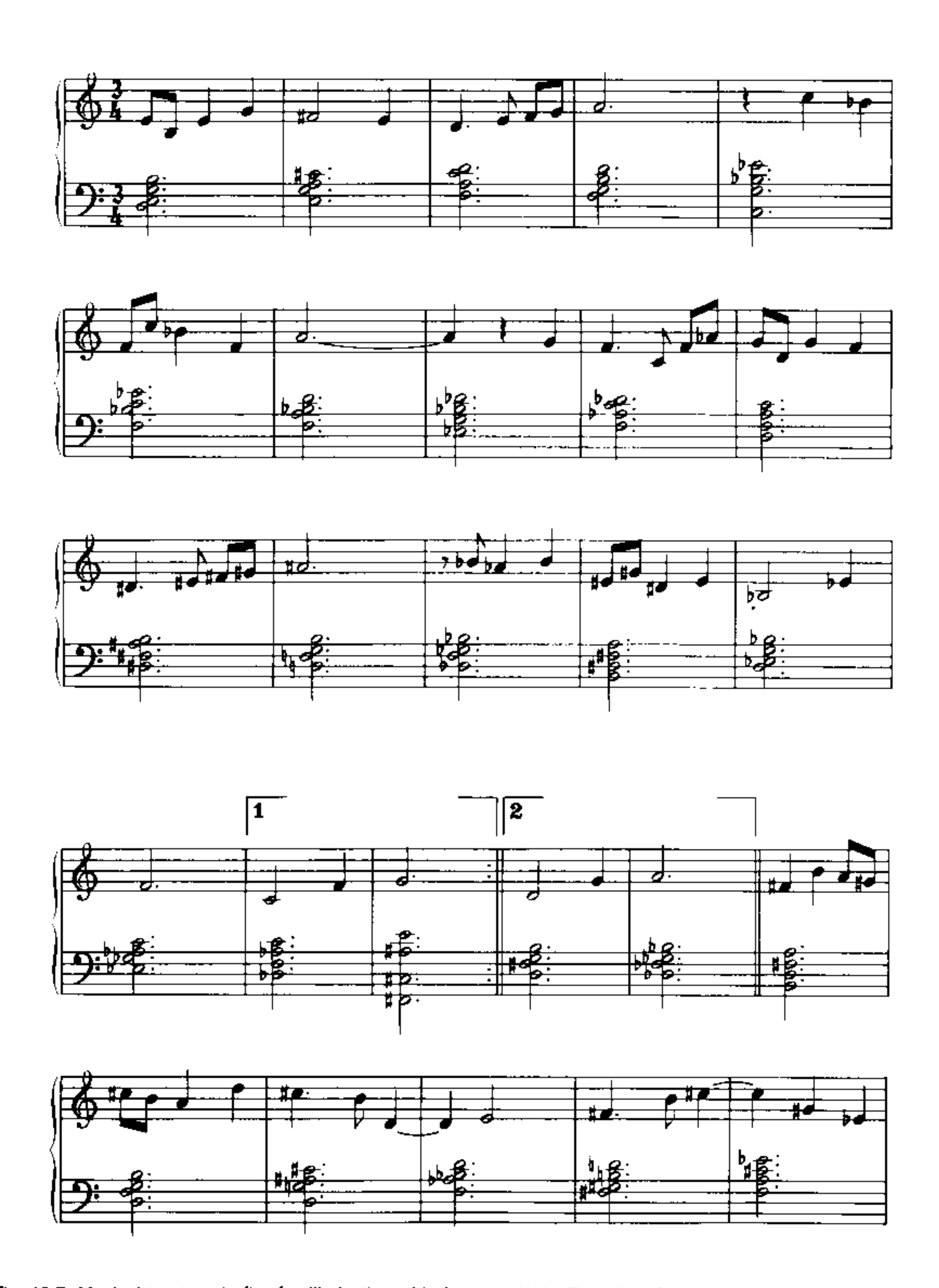


Fig. 10-7. Musical treatment after familiarization with the material in Fig. 10-6 (© 1978 Jaxitron).
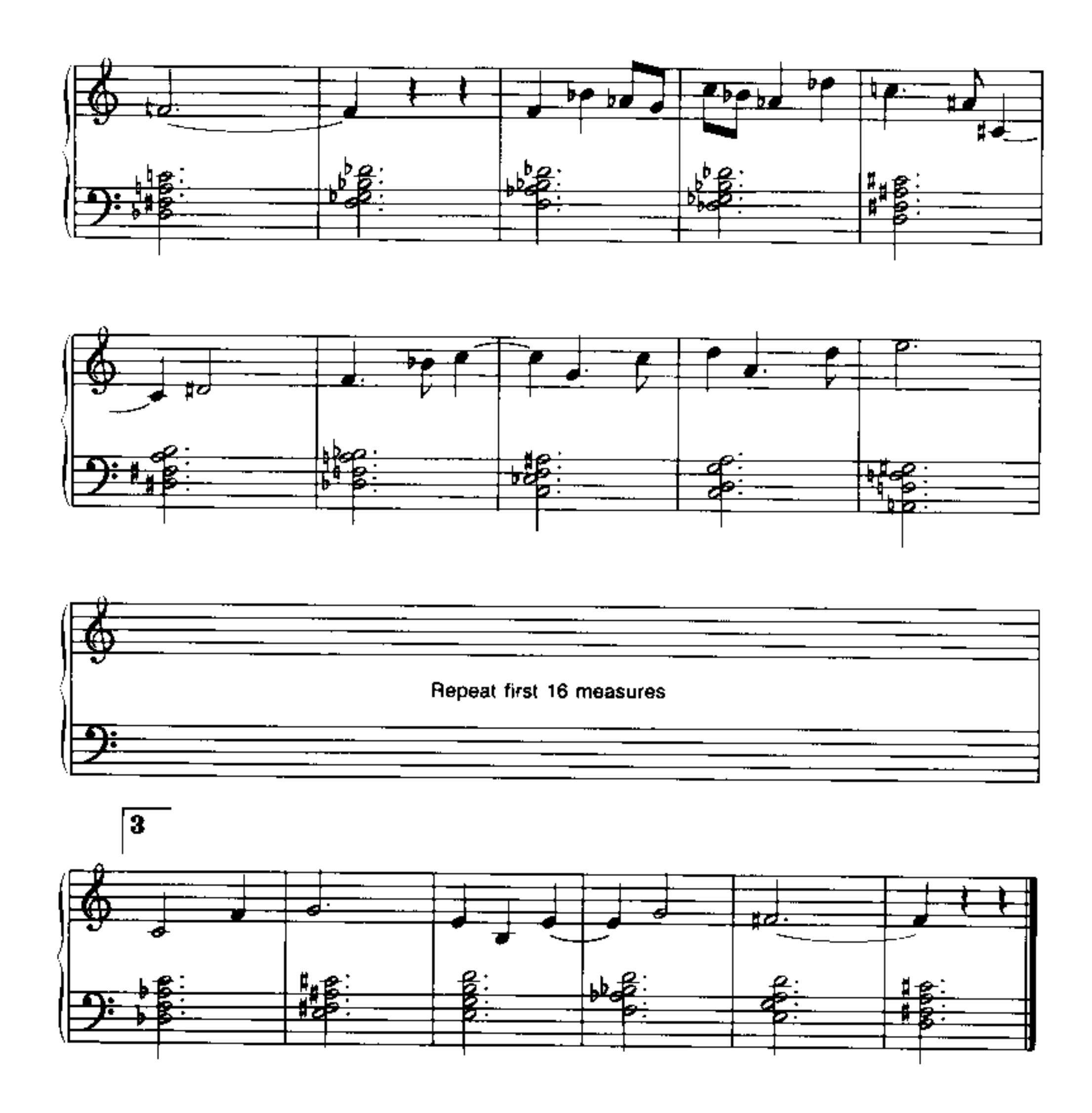


Fig. 10-7. Continued.

framework, the problems can be better defined than in the old. You can more readily sense the sharpening of judgement that comes with experience and not, as happens too often with conventional methods, fall into a style without conscious control. (Note how my inability to treat rhythm more objectively confirms this.) By altering a number here or a statement there, you learn to recognize the critical decisions that are the mileposts of style.

A naive remark by a mathematician or computer enthusiast can lead the less well-informed to believe that composition programs can be devised that will compose any kind of music and reduce the number of decisions required from the user to a most comfortable few. Let me reassure composers that a strong mathematical argument shows their profession to be secure—though it can't guarantee employment. The nature of the choices that specify
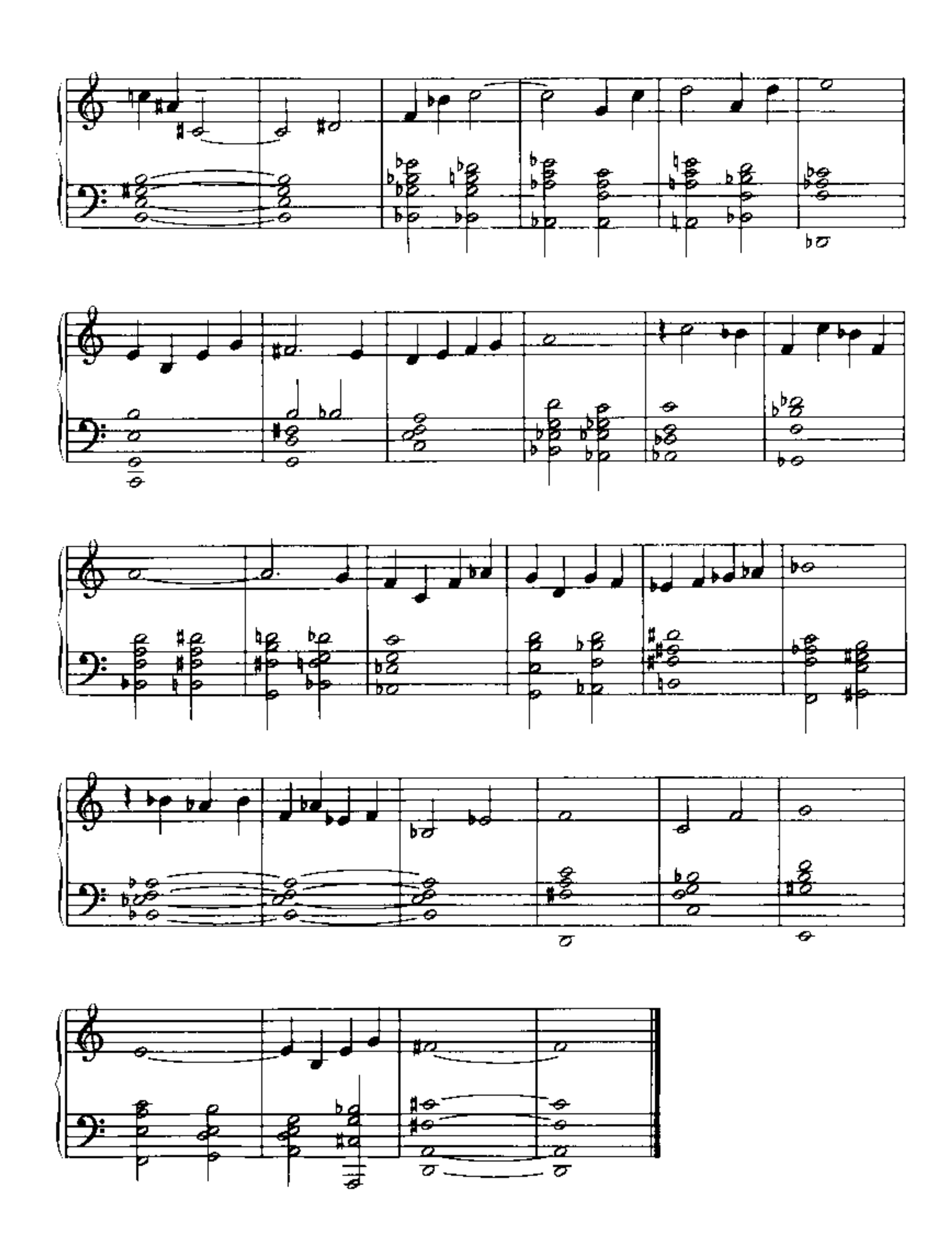


Fig. 10-8. Another treatment of the same material (© 1978 Jaxitron).
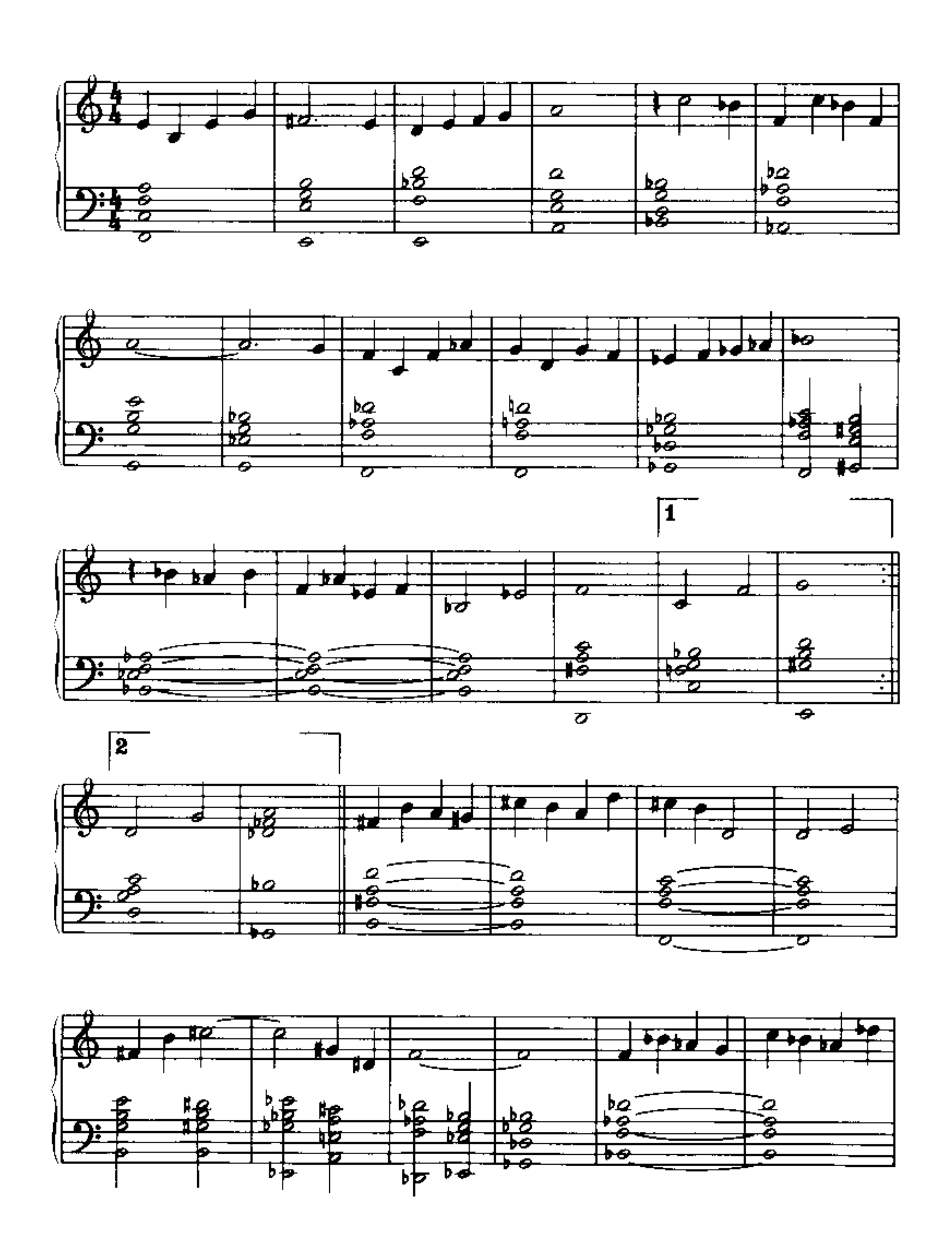


Fig. 10-8. Continued.

a piece of music can always be changed, but the number of choices cannot be reduced without restricting the "space" (idiom) in which the work exists. Special, highly stylized programs are indeed possible; general programs that can write any conceivable kind of music are not. But then, special, highly stylized composers exist; composers that can write any conceivable kind of music do not. And who, but such a composer, could even begin to place all of his subjective techniques into an objective framework—particularly the strict objective framework of a program?

Listing 10-1. Complete listing of a terminal session using FORM.

```
      FORM 'AABA'


ENTER A1


FORM 'ABCABD'


ENTER A1


KER←2 1 2
A1←(SCLI KER) MOTIF ¯3 3 2 ¯1


ENTER B1


B1←(SCLI 1⌽KER) MOTIF ¯1 1 1 1 1


ENTER C1


C1←1↓(SCL\¯1↑B1) MOTIF 2 ¯1 ¯3
C1←C1,C1,⌽¯3↑B1
INT A1
¯5 5 3 ¯1
NTRVL1←5
INT B1
¯2 2 1 2 2
NTRVL2←2


ENTER A2


A2←1↓+\(¯1↑C1),(INT A1),2↑INT A1
A1
0 ¯5 0 3 2
A2
¯4 1 4 3 ¯2 3
```

```
ENTER B2


B2←B1 XPOS 0,¯1↑C1


ENTER D1


D1←(12 SCLIT ¯1⌽+\KER) MOTIF ¯1,6⍴1 ¯2
D1←D1 XPOS 0,¯1↑B2
E←+\(¯1↑D1),NTRVL1,NTRVL2
LST3←¯1↓¯4↑D1
D1←D1,1↓E

NAME Z (=A1,B1,C1,A2,B2,D1)? AND/OR '→0'
FRSTA←Z
→0


ENTER A1


A1
0 ¯5 0 3 2
FRSTA
0 ¯5 0 3 2 0 ¯2 0 1 3 5 8 6 1 8 6 1 5 3 1 ¯4 1 4 3 ¯2 3 1 ¯1
1 2 4 6 6 4 6 1 4 ¯1 1 ¯6 ¯1 1
E←E+NTRVL2
E
¯4 1 3
A1←FRSTA,E
⎕
A1:
0:   C   ¯1G   +1C     D↑    D     C   ¯1A↑  +1C     C↑    D↑    F     G↑
     F↑    C↑    G↑    F↑    C↑    F     D↑    C↑  ¯1C↑  +1C↑   E     D↑
  ¯1A↑  +1D↑    C↑  ¯1B   +1C↑    D     E     F↑    F↑    E     F↑    C↑
     E   ¯1B   +1C↑  ¯1F↑    D   +1C↑  ¯1G↑  +1C↑    D↑


ENTER A2


E←E+NTRVL2
A2←FRSTA,E
⎕
A2:
0:   C   ¯1G   +1C     D↑    D     C   ¯1A↑  +1C     C↑    D↑    F     G↑
     F↑    C↑    G↑    F↑    C↑    F     D↑    C↑  ¯1C↑  +1C↑   E     D↑
  ¯1A↑  +1D↑    C↑  ¯1B   +1C↑    D     E     F↑    F↑    E     F↑    C↑
     E   ¯1B   +1C↑  ¯1F↑    B   +1C↑  ¯1A↑  +1D↑    F
)
FORM[26]
        )SAVE
```

```
09:13:24 06/08/78 MELCOMP
      →⎕LC


ENTER A2




ENTER B1


FORM 'XX'


ENTER X1


X1←(12 SCLIT KER) MOTIF (9⍴3 ¯1 ¯1),¯5 1
INT X1
5 ¯2 ¯1 5 ¯2 ¯2 5 ¯1 ¯2 ¯9 2
E←E XPOS ¯1,NTRVL1+1↑LST3
X1←X1 XPOS ¯1,(1↑E)-NTRVL2
INT (¯1↑A2),1↑X1
¯3
XX←X1
X1←XX,E,LST3


ENTER X2


X2←(XX,E) XPOS 0,¯1↑X1
¯1↑X2
8
X2←X2,E XPOS 1,¯1↑X2
¯1↑X2
10
X2←X2,E XPOS 1,¯1↑X2

NAME Z (=X1,X2)? AND/OR '→0'
BRDG←Z
→0


ENTER B1




ENTER A3


↑
```

```
ENTER B1


B1←BRDG
⎕
B1:
0:   D     G     F     E     A     G     F     A↑    A     G   ¯1A↑ +1C
     D     G     A     E   ¯1B   +1C↑   C↑    F↑    E     D↑    G↑    F↑
     E     A     G↑    F↑ ¯1A     B   +1C↑   F↑    G↑    D↑    G↑    A↑
     F     A↑ +1C


ENTER A3


A3←A1,5↑A1
⎕
A3:
0:   C   ¯1G   +1C     D↑    D     C   ¯1A↑ +1C     C↑    D↑    F     G↑
     F↑    C↑    G↑    F↑    C↑    F     D↑    C↑ ¯1G↑ +1C↑   E     D↑
  ¯1A↑ +1D↑    C↑ ¯1B   +1C↑   D     E     F↑    F↑    E     F↑    C↑
     E   ¯1B   +1C↑ ¯1F↑    B   +1C↑ ¯1G↑ +1C↑    D↑    C   ¯1G   +1C
     D↑    D

NAME Z (=A1,A2,B1,A3)? AND/OR '→0'
II←(A1,A2,B1,A3) FITRNG 7
II
4
N2P II+A1
0:   E   ¯1B   +1E     G     F↑    E     D     E     F     G     A   +1C
  ¯1A↑    F   +1C   ¯1A↑    F     A     G     F     C     F     G↑    G
     D     G     F     D↑    F     F↑    G↑    A↑    A↑    G↑    A↑    F
     G↑    D↑    F   ¯1A↑ +1D↑    F     C     F     G
N2P II+A2
0:   E   ¯1B   +1E     G     F↑    E     D     E     F     G     A   +1C
  ¯1A↑    F   +1C   ¯1A↑    F     A     G     F     C     F     G↑    G
     D     G     F     D↑    F     F↑    G↑    A↑    A↑    G↑    A↑    F
     G↑    D↑    F   ¯1A↑ +1D↑    F     D     G     A
N2P II+B1
0:   F↑    B     A     G↑ +1C↑ ¯1B     A   +1D     C↑ ¯1B     D     E
     F↑    B   +1C↑ ¯1G↑    D↑    F     F     A↑    G↑    G   +1C   ¯1A↑
     G↑ +1C↑    C   ¯1A↑    C↑    D↑    F     A↑ +1C   ¯1G   +1C     D
  ¯1A   +1D     E
N2P II+A3
0:   E   ¯1B   +1E     G     F↑    E     D     E     F     G     A   +1C
  ¯1A↑    F   +1C   ¯1A↑    F     A     G     F     C     F     G↑    G
     D     G     F     D↑    F     F↑    G↑    A↑    A↑    G↑    A↑    F
     G↑    D↑    F   ¯1A↑ +1D↑    F     C     F     G     E   ¯1B   +1E
     G     F↑
→
```

11 Harmony

Harmony seems, at first thought, to be a complex subject because it consists of simultaneous assemblages of pitches that change in time. Melody, consisting of only one pitch at a time, is certainly far simpler; one might even think of harmony as a combination of simultaneous melodies. But harmony is more than that, and its reason for being complex is not so simple.

In Western music, harmony is the fundamental principle that guides our perception of "sensibility" in pitch space. While harmony can be displayed as simultaneous groups of pitches called chords that are ordered in time, the pitches in a chord need not be played in simultaneity for us to perceive harmony. In fact, a familiar melody can be played without any form of accompaniment and its harmony can still be perceived—even "heard." When an amateur songwriter composes a melody "by ear," he is actually melodizing a harmony that he has in mind.

Perhaps I can clarify what I mean by our perception of sensibility in any kind of space through visual means. Figure 11-1 shows the well-known Necker cube illusion. It consists of three contiguous parallelograms in two dimensions. But our minds insist on finding an interpretation that agrees most "sensibly" with our experience, and our survival as a species is closely linked to the perception of two-dimensional images on our retinas as representations of objects in three-dimensional space. The guiding principle—the "harmony"—behind our perception of sensibility here is our innate sense of the rules of projective geometry. The lines we see instantaneously become the "edges" of a structure, but the angles between the various lines suggest two equally probably spatial configurations. Without any further clues to help us to decide whether the point that is common to all three parallelograms marks a receding or approaching terminus of the edges, two equally probable structures will "harmonize" the lines of the image. When you first look at the figure, you will find your mind involuntarily decides which of the two makes more sense, but then, after staring at

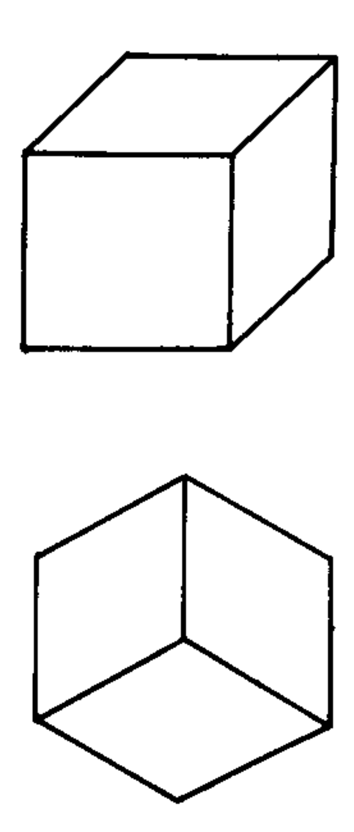


Fig. 11-1. The Necker-cube illusion.

it for a while, it will reverse its decision and uncontrollably switch back and forth between the two possibilities.

Figure 11-2 demonstrates that "correct harmonization" can be unique at every point in a figure so that, point by point, the figure makes sense, and yet, when taken as a whole, it may defy physical reality. The object depicted here almost portrays the temporal nature of a melody. As you follow it around in its course, you continually sense where it has been and where it's going. Your mind only balks when you try to match the figure in this space with the projection of something in physical space.

Reversing the analogy, we might say that a melody is harmonically sensible if its "line" of successive tones serves as a projection in a one-dimensional pitch space of the outline of a structure in a space with more dimensions. For some reason, our minds require that this larger space exists and that melodies be projections of structures in that space. Abstract analogies aside, we expect the notes of a melody to suggest particular chords, and we also expect the succession of notes in the melody to suggest particular sequences of chords. Harmony has certain "sensibility" requirements: Each chord must, by itself, be a "sensible" structure and successive chords must lead "sensibly" from one to another. In combination, melody and harmony interact so that the individual sensibility requirements of one component can reduce the number of possibilities available for "choosing" the other.

At least, it should now be clear why harmony is a complex subject. In its conventional study, one memorizes a large number of so-called rules about certain chord structures: how they are built in terms of scales or intervals, how a semiconsistent terminology is used to describe them, and under what conditions one chord may follow or lead to another. The question of sensibility is not addressed except to say that there will be no problem if you follow the rules, even though professional composers consistently break the rules and yet maintain sensibility. If computers are to help us with harmony, we must be able to write algorithms that meet sensibility requirements—so a conventional approach will be of little value.

We'll begin with a brief introduction to the most common aspects of harmony. Then, without getting into the usual details of the subject, we'll examine some rather abstract ideas which lead to a hypothesis that accounts for all the standard harmonic structures and allows algorithmic approaches that ensure sensibility. We'll go into the temporal problems that arise in combining harmony and melody in the following chapters.

DEFINITIONS AND TERMINOLOGY FOR CHORDS

The term *chord* will be used most unpedantically to describe either an absolute or relative pitch structure. In the absolute sense, a chord has

a specific root tone to which an interval sequence is added (literally!). The interval sequence is itself a chord in the relative sense. For example, the interval sequence 4 3 describes a major triad, but, when added to the root tone, zero, it produces the three-note chord

$$0, 0+4, 0+4+3 = 0\,4\,7 \text{ or } \mathrm{C\,E\,G}$$

which is a C-major triad. Reversing the interval sequence to 3 4 and changing the root tone to two gives us

$$2, 2+3, 2+3+4 \quad = \quad 2\,5\,9 \quad \text{or} \quad \mathrm{D\,F\,A}$$

which is a D-minor triad.

By combining each of the twelve tones individually as a root tone with the two sets of intervals, 3 4 and 4 3, we can build all possible minor and major triads by the process just shown. The general algorithmic form for the process would be

```
CH ← +\RT, IS
```

where RT is the root tone and IS the interval sequence needed to produce the chord CH. To generalize further, any pair of intervals in IS will produce a triad, any triplet of intervals will build a tetrad or four-note chord, and so on. However, you will find something unpleasant or strange or dissonant about most such structures. The reason behind this—or better, a hypothesis for the reason—will be fundamental to all that follows.

But first I must introduce certain forms of nomenclature. It is most convenient, even necessary, to be able to name a particular kind of chord or to refer to a particular tone in a chord with the smallest number of words that will convey the most information. A programmer will naturally prefer to use the "ordinary" indices of the notes in the pitch structure (0, 1, 2, etc.). Oddly, the musician addresses the same tones by the odd ordinals: first, third, fifth, and so on. This is based on the classical approach to chord formation in which you can think of a scale in its first expansion or simply say that chords are to be built "in thirds." The first expansion of C D E F G A B (C) would be C E G B D F A (C), and the tones are addressed as 1 3 5 7 9 11 13 (1).

From force of habit, and also because musicians already have a set of adjectives available for expressing tonality, I will most often use the musical approach to terminology. In a major triad, such as C E G, the consecutive tones are usually called so:

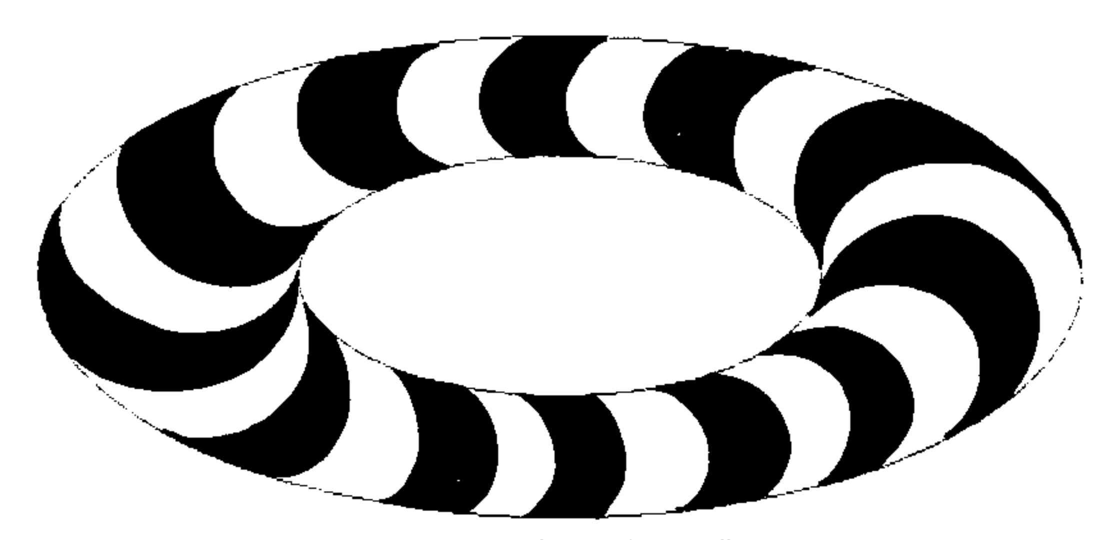


Fig. 11-2. An illusion based on the way our senses try to interpret "bending" and "roundness" from visual information.

C	index 0	first, tonic or root
E	index 1	major third
G	index 2	fifth

The adjectives used (in this case, the only one is "major") are the same as those given earlier in discussing intervals: major, minor, augmented, and diminished.

There are at least two reasons for saying these adjectives are only fairly meaningful. The first goes past the adjectives to the ordinal numbers. In dealing with chords like C F B that are not clearly built in thirds, descriptions such as tonic, augmented third, and quadruply (quadrupally?) augmented fifth make the adjectives less useful than expletives (I know how to spell most expletives). The second reason applies even when chords are built in thirds; there is no systematic set of rules that determine which tones require the adjectives. Major and minor triads are distinguished by the interval between the first (0) and the third (1) of the chord, that is, a minor triad has a minor third where the major triad has a major third. But augmented and diminished triads are named for the interval between the root and fifth, and everyone is expected to know that the augmented triad has a major third while the diminished triad has a minor third. Things get so bad when we try to name four-note chords that consistency and convenience are both lost in names such as "minor triad with a major seventh" or "major seventh chord with a minor third," both referring to the same four-note structure whose intervals are 3 4 4.

There are handicaps here for both musicians and programmers. While the musician can rely on experience to select chord structures, he will rarely choose a structure he cannot name. The programmer has greater freedom of choice, but no experience to rely on for those important passages that call for the chord he "hears" but doesn't recognize in any form of nomenclature. In this and the following chapters, ideas and methods will be presented that should be helpful to both, though I will not muster enough intestinal fortitude to suggest a self-consistent naming convention.

THE CONCEPT OF ORDER

The axiomatic assumption here will be that sensibility in everything we observe depends on our recognition of order. It may be that experience is more fundamental than order in that we learn to recognize the order that exists in the things we observe. But the assumption here turns that around just a bit so as to say that there may be orderings that will immediately "make sense" even without prior experience. With experience we also learn to make sense out of particular orderings that might at first appear chaotic.

At the most basic level of observation, we subconsciously distinguish between events governed by natural randomness and "intelligent" guidance. If all the ink molecules on this page had been positioned in a random way, you would imagine some accident in the printing process had caused it. If they had somehow managed to fall into a geometric arrangement, you'd wonder how this "figure" fits into the discussion. Had I written the page by rolling a basketball over my keyboard, it would be recognized immediately as a page of text, but a moment later as gibberish. Without engaging consciously in any sort of calculations, each of us, when confronted by a collection of colors, shapes, and sounds, decides almost instantaneously whether any effort or special circumstance was responsible for its creation. If any degree of "contrivance" is apparent, our attention focuses on it as long as is required to understand or appreciate the order it contains. With experience, we develop successively higher levels of discrimination that let us decode information carried by order and appreciate styles of order. It should be obvious that such universal behavior in a species must serve an evolutionary function. Though we can't go into it more deeply here, that makes it worth mentioning that esthetics in particular and the arts in general are of more importance than our culturally based attitudes encourage us to believe!

In the case of melody, we saw that sensibility was largely dependent on creating order in time. In the time dimension, the perception of order must

rely on memory, so methods designed around the use of repetition seemed obvious candidates for the creation of ordered patterns. Sensibility related to the pitch dimension was lightly passed over by saying that it depended on something called tonality. We are now at the point where we must immerse ourselves into that dimension in order to "see" just what might constitute order and how we perceive gradations in that order.

ORDERING TRIADS IN TWELVE-TONE TUNING

Symmetry is the most obvious form of order in certain "spaces." The notion that anything resembling a circle or equilateral triangle might be observed in pitch space seems rather absurd at first. But, given the cyclic nature of the twelve pitch names and the way we sense the twelve tones regardless of their octave, let's view them arranged on the face of a clock—ironically, an analog clock (Fig. 11-3). Pitch numbers are already in place, providing we let the value twelve represent both zero and the octave. Although pitch names will be shown adjacent to the numbers here, it would be better to consider them to be a sample representation and say that, in general, zero will always represent a root tone on this diagram, not necessarily the pitch C.

The most symmetric triad is represented in the figure. Its symmetry is apparent numerically if, instead of the pitch numbers, we note the intervals separating the cyclic arrangement that begins and ends on twelve. We see three equal intervals of four semitones (4 4 4). Notice, there are no permutations of this set of intervals; the structure (an augmented triad) is unique in its symmetry class.

Next, we'll examine all possible ways to "break" the symmetry and still retain three pitches and three intervals on our clock, always beginning and ending at twleve. We will assume that some degree of disorder can be associated with the "size of the fracture." The restrictions stated above require that the sum of the intervals remain equal to twelve. The class of structures "nearest" to the

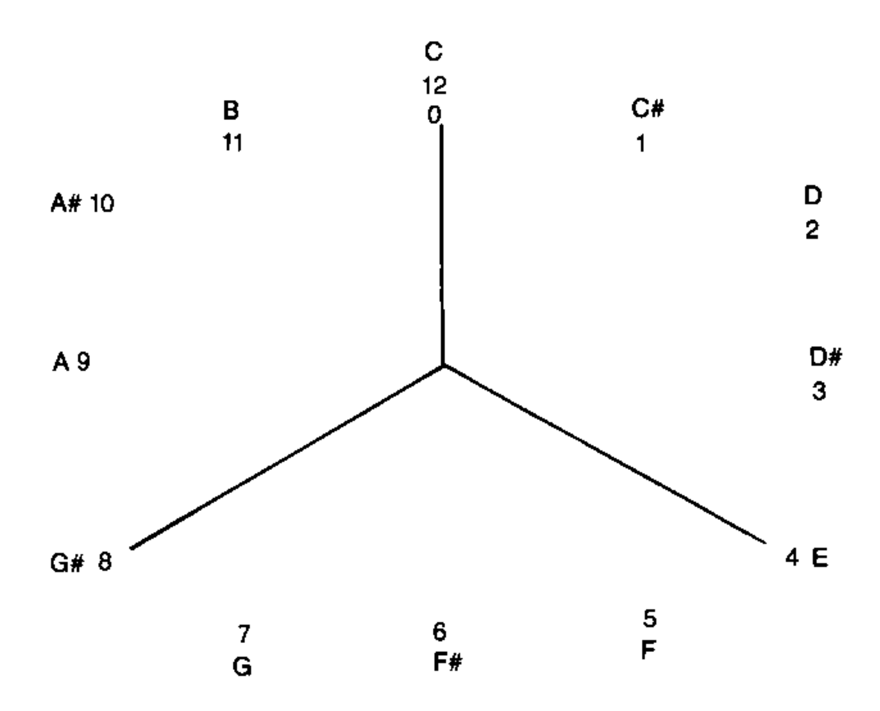


Fig. 11-3. The cyclic symmetry of the twelve pitch names.

totally symmetric one would then deviate minimally from the intervals 4 4 4. The smallest change that can be made would add plus one to one of the fours and add negative one to another. Specifically, adding ⁻1 0 1 to 4 4 4 gives us the intervals, 3 4 5. Now we can either permute this first set of intervals or permute the elements of the distortion (⁻1 0 1) and then carry out the addition to find all other structures that have the same amount of disorder. Better still, we can let the computer do it as shown in Table 11-1 at the end of the chapter. Notice that the pitches corresponding to this "least" distortion produce all six possible settings (inversions) of the major and minor triads.

The algorithm used to create Table 11-1 established ⁻1 ⁻1 2 as the next smallest distortion. This produces a group of three structures (through permutation) showing the three possible settings of a diminished triad. Notice the second member of this group is itself symmetric: the distortion is ⁻1 2 ⁻1, the intervals are 3 6 3. Before examining the next class in the table, note there are nine structures for which the smallest interval is three. Groups of structures are clearly separated in the table. We will use the word *class* to include all groups with the same smallest interval.

The next class begins with the distortion ⁻2 1 1 and produces the three settings of a triad built in perfect fourths. The minimum interval in these structures is two. When the classical masters of composition worked their way down to this set of structures, classical theory accounted for them with such terms as *suspension* and *appoggiatura*. But the fact that they appeared in the same era as did structures in the previous group is worth noting along with the observation the ⁻1 ⁻1 2 and ⁻2 1 1 have equal symmetry!

Continuing down the list, we find three more groups with a smallest interval of two semitones. In all, we find there are 18 structures for which this is the minimum interval. Their musical flavor tends toward the "ordinary" seventh and minor seventh chords. The remaining structures take us farther from the perfect 4 4 4 symmetry, completing the table with 27 structures having minimum intervals of one semitone.

ORDER, SYMMETRY, NUMBERS, AND "NORMAL" ESTHETIC REACTIONS

There are two ways of disliking art; one is to dislike it; the other is to like it rationally.

—Oscar Wilde

Perfect symmetry strikes us as intellectually interesting though not emotionally noteworthy. Whenever a fixed number of items may appear in a given "space," there will be a relatively small number of ways to arrange them so as to achieve a perfectly symmetric configuration. The smallest deviations from such order arise in a large number of ways, yet this number is still small compared with the number of all possible arrangements. With increasing disorder, we find increasing numbers of arrangements. In fact, the latter is a measure of the former; there are many more ways to create disorder than there are to create order. Even in the case shown for triads where the total number of possible configurations is only 55, the class structure breaks down as follows:

minimal interval	4	3	2	1
number of structures	1	9	18	27

Esthetically, in our first encounters with a particular art form we seek out the challenge presented by the class of arrangements with minimal disorder—the challenge being to recognize the features of each individual arrangement that make it differ from "perfection" and from the other members of its class. Esthetic enjoyment accompanies success in meeting the challenge. With experience, the challenge disappears and we graduate to the next higher class of disordered arrangements. This class generally has a far greater number of elements, so that it will be more unpleasant to undergraduates but more challenging to graduates of the first class. Actually, the process is more continuous. As one becomes familiar with one class, though not necessarily with all members in that class, certain members of the next class begin to be recognized, and as one adjusts to the more complex class, the previous class loses its appeal and becomes trite. The more members a class has, the

harder it is for an observer to single out one or a few of those members in order to perceive that first spark of recognition needed for esthetic growth. So the larger the class, the more chaotic it appears to the uninitiated.

These remarks carry over from "normal" individual reactions to entire cultures through the behavior of large numbers of individuals. The list of triads just shown (Table 11-1) confirms this historically. Of course, our culture outgrew triads before getting halfway through that maze of classes.

ORDERING TETRADS

Exactly the same kind of analysis can be carried out for four-note chords (Table 11-2). In this case we find:

minimum interval	3	2	1
number of structures	1	34	130

Our starting point for "perfect symmetry" here is the interval structure

3 3 3 3.

The first entry in the rightmost column in this table shows the symbolic representation I chose for this diminished seventh chord.

The class with minimal distortion is formed by adding the intervals

⁻1 0 0 1.

This creates the group of twelve structures beginning with the intervals

2 3 3 4.

This group contains the various inversions of the ordinary seventh chord, the minor seventh, and the "small" seventh:

C	E		B♭	=	C 7	(seventh)
C	E♭	G	B♭	=	C MI7	(minor seventh)
C	E♭	G♭	B♭	=	C – MI7	(small seventh)

The three groups of 22 structures that follow contain the more unusual sounds in this class. Many composers have used them, but these structures have "slipped through the cracks" of conventional music theory because they are not built in thirds.
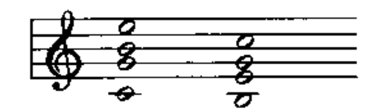
The first group of structures in the final class opened composers' ears to the use of semitones and the major seventh interval in tetrads. These twelve structures contain all settings of the major, minor and augmented triads combined with the major seventh:

C	E	G	B	=	C MA
C	E♭	G	B	=	C MI
C	E	G♯	B	=	C +

THE CONCEPT OF TENSION

A satisfying insight seems to be the most we can claim from the approach just demonstrated. That insight will be of value as we continue, but the approach itself doesn't have the power we need. It does not account for structures containing two tones or more than four. Expanding the clock face to 24 hours might help, but that contradicts the psychological effect of the single octave and so would introduce artifacts of method into the results. For that matter, there is an annoying artifice in the given approach.

By closing the octave in a full circle, we make apparent the symmetry properties of pitch space. As a result, chords are indeed collected persuasively by type, group, and class. However, in practice we seldom close the circle; chords are not generally used with the lowest pitch duplicated one octave above. Without that duplication, chords in the same group—even of the same type in that group—may seem unrelated! For example, there are four settings of a major seventh chord in one octave as shown in the table. But in "calculating" four-part harmony that is not restricted to one octave, a program might construct a C-major seventh chord in any of 24 different settings. Here are just two possibilities:



The second one would be as out of place in Debussy's "Reverie" as would be the first in Berg's "Lyric Suite." Obviously we need deeper understanding of the situation than can be provided by the concept of order alone.

Schillinger used the word *tension* to express the psychological dissonance in a chord. He even used it in a relative sense to imply a difference in tension between one chord and another. Sadly, I could find no indication in his work as to how one might begin to compute tension. In fact, at one point, he equated a linear scale of tension with successive tones in a harmonic structure, so that his results led to a contradiction. But his choice of the word not only seems appropriate, it suggests physical ideas that enhance the notion of structures in pitch space. Our structures now seem to possess forces (tension or compression) between their parts and potential energy due to these forces.

To say that order is the mark of effort or intelligence is to recognize that energy must be spent (work must be done) to create order. Our application of distortions to break down the perfect symmetry of the augmented triad or the diminished-seventh tetrad is quite analogous to "nature's way" of adding thermal, chemical, or radiant energy to make ordered physical systems increasingly disordered. Enclosing our system within the walls of an octave extends the analogy to a closed physical system in which the walls also share the energy. So those distortions, although numerical, do not measure the energy of the "free" structure. But it is obvious to anyone who listens to the chords in Tables 11-1 and 11-2 that tension increases with increasing disorder.

Quantifying Tension

Our investigation of order has yielded some important clues for establishing a numerical scale of tension based on intervals. We can let the totally symmetric structures establish a "reference level" and say they have no tension. This indicates the intervals 3 and 4 are without tension. Then, moving down the tables to structures with increasing disorder, we can say intervals of 2 and 1 have increasing tension. We can also note that other intervals appear in these chords between notes that are not adjacent. That is, the intervals 3, 4, 6, 8, and 9, as well as the octave, appear in the structures that have no tension. Then with successively increasing tension, we find intervals of 5 and 7, 2 and 10, and finally 1 and 11.

Subjectively I have no difficulty placing the last four intervals in a specific order—10, 2, 11, 1 with tension increasing. But my discrimination of tension is not sensitive enough to differentiate between 5 and 7 or, for that matter, between these musical fourths and fifths or any of the tensionless intervals. I tried to overcome my insensitivity by prolonged musical abstinence, but quickly found there was no way to do this in my regular, day-to-day environment. In a thoroughly unscientific sampling, I found a few people with no musical background (some unable to carry a tune) who could rank the more consonant intervals but found all seconds and sevenths too tense to separate. Unfortunately, some found tension in places where there should be none (the octave!), and the orderings of different individuals were completely inconsistent. On the other hand, all my musical cohorts found distinguishable tension *only* in seconds and sevenths (the chromatic intervals 1, 2, 10, and 11) and all ranked them in the same order that I did.

There was also agreement among the musicians that tension values need to be well-dispersed; a chord containing a number of the less tense intervals has lower overall tension than one containing a single interval that is higher in the tension scale. This suggests that our perception of tension, like our perception of pitch and volume, is logarithmic.

One further consideration remains before we can assign values. How is tension affected when octave separations are added to the psychological interval between tones? This turns out to be rather complex, but the effect is small. In general, tension seems to decrease (logarithmically) as more octaves are added. That is, the interval 10 has just a bit more tension than the interval 22 (10 plus 12), but seems to have even more tension than the interval 34 and far more than 46. The same effect is found

for the psychological intervals 2 and 11. A few of the musicians say this also applies to the interval 1, but most of them find the interval 13 notably more tense than the minor second! I believe the reason for the disagreement lies in their relative experiences. Certain idioms (including MT) use the interval 1 far more than 13, while others use them equally (a lot or not at all).

Because the effects noted above are small and chords usually have limited range, it seems reasonable to ignore the effect of additional octave separations for the present. That is, we'll consider the interval from C upward to D to be a major second (chromatic value 2) regardless of the number of intervening octaves. Further, we'll select a scale of values that will allow a chord with any (reasonable) number of intervals of a certain tension to be less tense in the aggregate than a chord with a single interval having the next tension value in the scale.

Let's begin then by adopting the logarithmic scale

··· .001 .01 .1 1 10 100 1000 10,000 ···

Now, if we assume only four intervals require nonzero values of tension, it will be most convenient to assign tensions of 1, 10, 100 and 1000 to the minor seventh (10), major second (2), major seventh (11), and minor second (1), respectively. If any inconsistencies become apparent, it should not be hard to find which of our assumptions are inadequate and make changes accordingly. For example, should less tense intervals need nonzero values, they can be spread appropriately among the (infinite number of) points in the scale between zero and one. Should the assumption of octave equivalence be found deficient, we can insert or shift values as needed to account for our perception. And, if three intervals of a particular tension seem about equal to a single interval of higher tension, we might recalibrate our scale to powers of three instead of ten:

···.11111 .33333 1 3 9 27 81 ···

Calculating Tension in a Structure

Without applying any refinements at all, this relatively simple scheme works very well. Large catalogs of structures can be sorted according to their calculated tensions; not only will people agree on the ordering found, but there is even chronological agreement with historical usage—implying that cultural maturity follows the path of increasing tension, as might be expected. Instances do arise where certain chords in a catalog seem out of place, but I believe this is more the result of conventional usage than the need for applying any of the refinements.

The total tension in a chord is calculated as the sum of the tensions of all the intervals in the chord. This simple algorithm cuts directly through the inextricably knotted rules of conventional harmonic theory. It lets you measure the effect of adding a tone (chordal or melodic) to any given structure. For any particular chord, we can construct a 12-element vector that shows the tension of every pitch with respect to the given chord. For a major triad, the vector would be

0 1000 11 100 0 1001 110 0 1000 10 1 100

This is more easily seen in a keyboard-like diagram, reflecting the black-and-white pattern of the notes; The diagram for a C-major triad is shown in Fig. 11-4.

To see, for example, why the note D has a tension value of 11 with respect to the chord, we assume it is added *above* all the chordal tones and ignore intervening octaves. Then, its interval from C is two (tension = 10), it is ten semitones above E (tension = 1), and seven semitones above G (tension = 0). The sum of these tensions (10 + 1 + 0) gives the value shown in the diagram.

Similar diagrams are interesting, instructive, and useful in analyzing the relationship of every pitch to any given chord structure. Just remember, in tabulating the intervals, one must consider the given chordal tones to lie in a lower octave. (An APL function for the calculations will be shown below.) For C-minor, augmented, and diminished triads, the diagrams in Fig. 11-5 can be drawn.

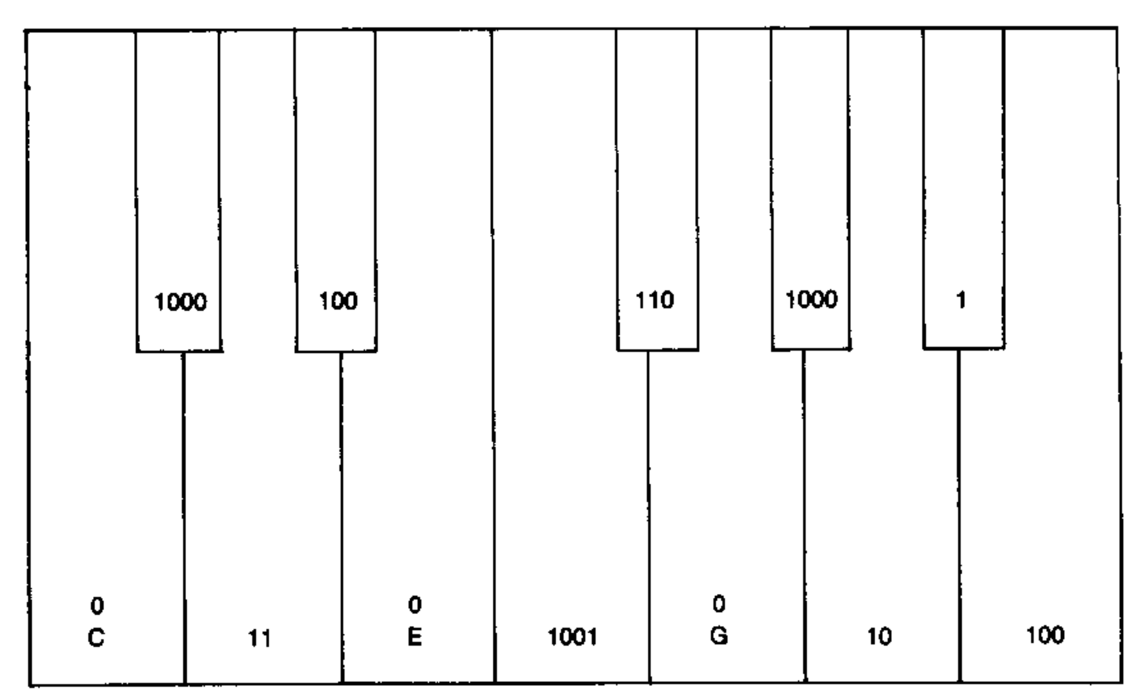


Fig. 11-4. Tension of each with respect to a C-major triad.

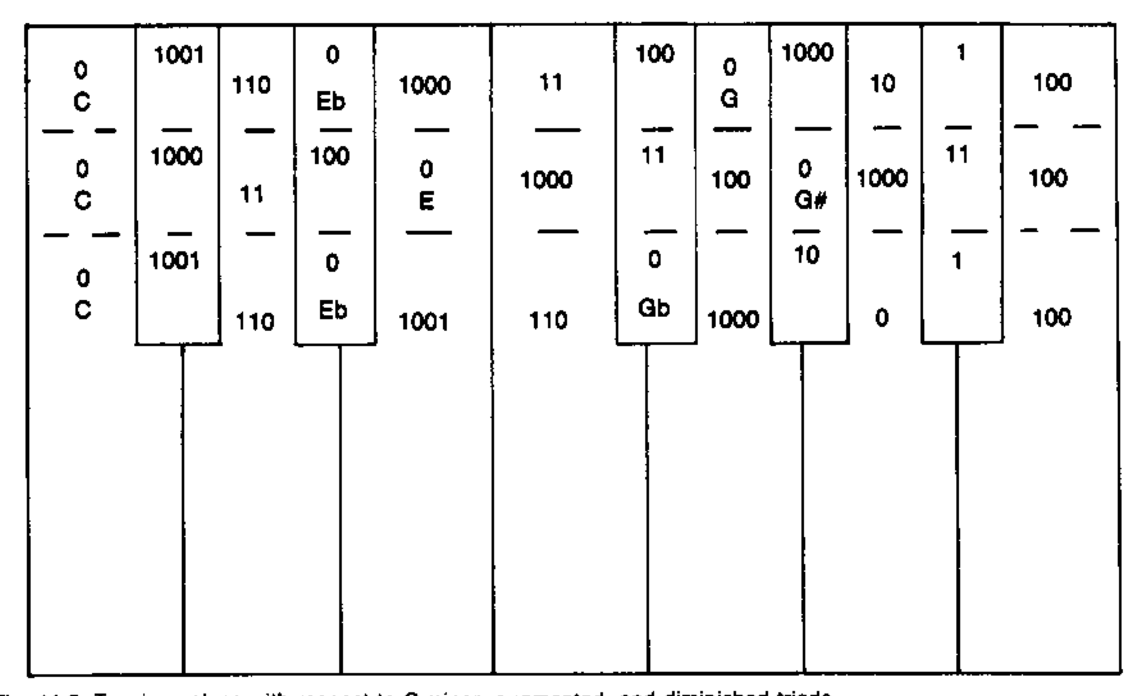


Fig. 11-5. Tension values with respect to C-minor, augmented, and diminished triads.

```
      ∇ Z←TNSN V
[1]     Z←10⊥+/(1 11 2 10)∘.=TONSYS|(,(⍳⍴V)∘.>⍳⍴V)/,V∘.-V
      ∇
```

Fig. 11-6. Function TNSN calculates the tension in pitch vector V.

Tension with Respect to a Structure

We can draw the same kind of diagram for a chord containing any less orthodox set of pitches, e.g., C, F, and B. But note that this chord has tension of its own, while the triads used above did not. So first we must examine the calculation of tension more closely.

To clarify, consider the one-line algorithm listed in Fig. 11-6. The right argument for this function is a vector of pitch numbers whose tension is to be evaluated. Its entries, from left to right, should represent the notes in a structure ordered from lowest pitch to highest. To find the tension in the chord (C F B), we could write

TNSN 0 5 11.

The algorithm would return the *total* tension in the structure (as a scalar value). This means that if we want the tension of D♭ with respect to this chord, we cannot simply write

TNSN 0 5 11 1

because this would return the tension of the entire four-note structure. Instead we must use

(TNSN 0 5 11 1) – TNSN 0 5 11

which effectively calculates the tension in the tetrad and then subtracts out the chordal tension contributed by the triad. Doing this for each of the pitches in the octave individually would then let us draw the diagram shown in Fig. 11-7.

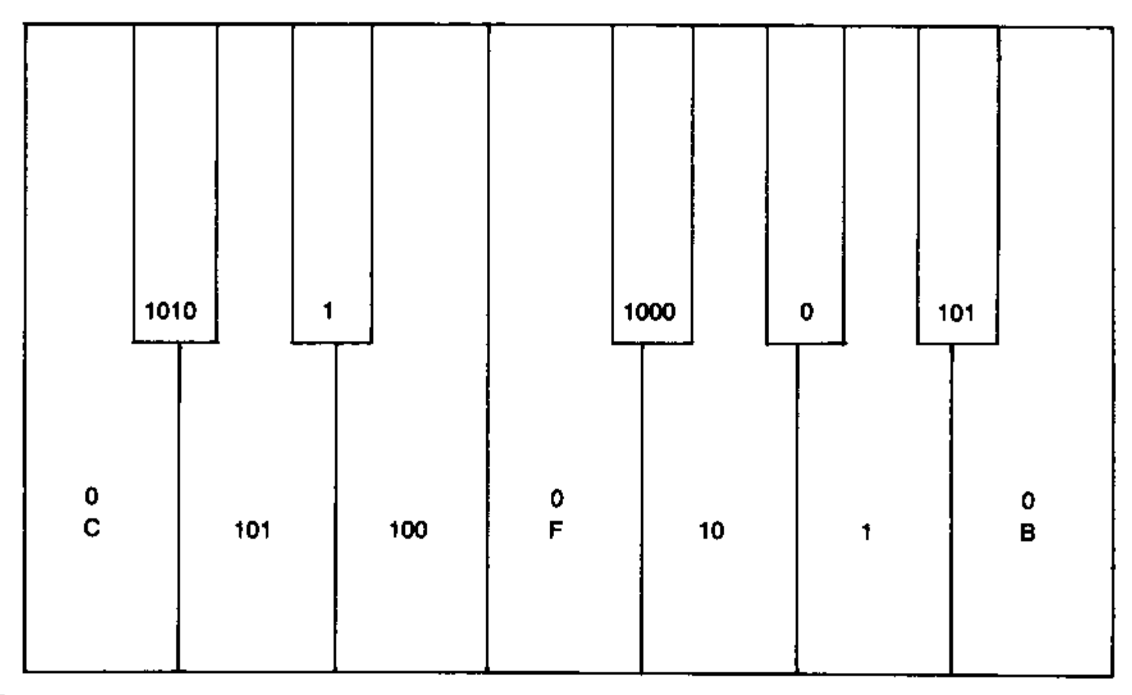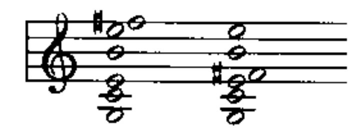


Fig. 11-7. Tension with respect to a CFB triad, with values produced by the TNSN function.

In voicing a chord for orchestration, a related problem arises. A structure may contain repeated pitches in different octaves. We have already seen that the position of a tone in a structure can affect the tension. Repeating a pitch in a different octave will do the same thing. Further, different tone colors create psychological complexities that confuse our perception of the way relative octave positions of different instruments affect tension. The simplest approach to this problem is to ignore any pitch in the structure that also appears in a lower octave. The function **UTNSN** will do this, using the **UNIQ** function. Both are listed in Fig. 11-8.

Now, the two chords shown here



would have tensions of 220 and 211, respectively, if the **TNSN** function were used for the calculation. **UTNSN** treats the repeated pitch, E, in both cases as an "overtone" that doesn't contribute to the tension, so both structures have tension values of 210 based on their unique pitches.

TONALITY, ORDER, AND TENSION

Schillinger pointed out the tendency in our culture to connect melody and harmony through seven-note scales. He went out of his way to show all 36 seven-note structures built in thirds, but he ordered them according to some mechanical permutation scheme that served no useful purpose. We can now reassemble the list in order of increasing tension, showing the intervals, pitches (taking C as the root tone), and tension of each structure on one line of a table. The structures are sorted and grouped by tension in Table 11-3 at the end of the chapter.

The involuntary search by our minds for order in pitch space persists in the time direction. Upon hearing an unaccompanied melodic motif, we seek out patterns in the successive notes that suggest an underlying structure in the pitch direction. If tension were consistently analogous to the forces responsible for potential energy in a structure and we automatically looked for realistic physical behavior in the things we observe, then it would make sense for our minds to look for a harmonic structure that minimizes the net tension of all the melodic tones in the motif. This will in fact turn out to be the case, but there is an additional force between melodic tones that relates to and "extends" the harmonic structure—tonality.

Static structure (harmony) seem most stable with no tension between adjacent tones. We expect highly ordered harmonic structures to be built in

```
    ∇ Z←UTNSN V;K
[1]   Z←10⊥+/(1 11 2 10)∘.=TONSYS|(,(⍳⍴K)∘.>⍳⍴K)/,K∘.-K←UNIQ V
    ∇

    ∇ Z←UNIQ V
[1]   Z←((⍳⍴V)∊V⍳V)/V←,V
    ∇
```

Fig. 11-8. Function UTNSN calculates the tension in V, considering only the first occurrence of each unique pitch. Function UNIQ returns the unique entries in vector V.

thirds, and voiced chords are derived from these structures. A pitch moving in time (melody) seems to require a "continuous" path in pitch space. We expect scales to be built in seconds, and melodic motifs are derived from particular scales. On expanding a scale into one of the above structures, its "flavor" or tonality carries over. Tonality is a characteristic of a "complete" (seven-part) structure, whether the structure is used melodically or harmonically.

As a result of this analysis, there are two different approaches we can take with regard to tension in composing melody and harmony. The diagrams shown earlier, giving the tension of each pitch relative to a given chord, provide one approach—i.e., the chord may have tension and the melody may carry tension with respect to the chord. The other method is to rely on tonality. Using one of the seven-part structures, a chord can be chosen from the lower tones while the melody ranges over any part of the associated scale. The melody can then have considerable variation in tension, through the net tension cannot exceed that of the entire seven-part structure.

The seven-part structures shown in Table 11-3 were restricted to those that could be formed using only major and minor thirds between successive tones without repeating any pitch. Table 11-4 shows a list of structures that are not quite so restricted but which are applicable to the MT idiom. For each structure, you see the total tension, the interval sequence, the corresponding pitch names if C is taken as the root, and an entry labeled "class". The structures are ordered and grouped here according to this last heading. The entries in that last column show my own terminology for describing the first four notes of each structure. The structures within each group are ordered according to their interval sequences.

MORE COMPLEX STRUCTURES

We're now capable of deriving some fascinating harmonic structures and evaluating their "usefulness" in terms of tension. Using Schillinger's concept of strata harmony, a function was written to create various structures in different strata and combine the strata through different intervals. The lowest stratum always has its structure built on the pitch C. Specifying two strata, each containing the same structure (a major triad) and requesting the two strata be separated in all twelve possible ways, gives us Table 11-5. On each line we see the pitches in each stratum, the number of unique tones in the entire structure, and the tension of the whole as computed by the algorithm, **UTNSN**.

In the first of two more brief examples (Table 11-6), a triad containing two perfect fourths was used in the same way that the major triad was used in the previous case. And in the second of these (Table 11-7), the same triad appears in the lower stratum while a structure containing a whole tone and a half tone occupies the upper one.

A more ambitious catalog was produced by allowing both strata to contain any of the triads formed from the following pairs of chromatic intervals:

4 3
3 4
4 4
3 3
5 5
5 6
6 5

With seven possible forms in two strata and twelve choices for the interval of separation, $7 \times 7 \times 12 = 588$ structures are found, as shown in Table 11-8. The last two columns again show the number of unique pitches and the tension.

When such lists are compiled without considerations of tension and the sorting these considerations make possible, they impress the would-be composer as relatively useless because most of the structures found "just sound terrible." With the use of tension and sorting, one can dive right in to one's preferred depth. While such preference is highly subjective, students of composition generally divide structures into four populations according to tension and assign

characteristics of their own, such as:

Tension range

	< 10	trite, simple, blah
10 to	< 100	pretty, lush, schmaltzy
100 to	< 1000	interesting, sonorous, wild
1000 and	up	complex, strange, oops

My computer tells me there are 17 structures in the above list that fall into the first category, 22 in the next, 227 in the third, and 322 in the last. Such statistics demonstrate the value of the concept of tension and the consistency of the hypothesis on which it is based.

Table 11-1. Structures Formed through Successively Greater Distortions of Threefold Symmetry in the 12-Tone Pitch System.

DISTORT			INTERVALS			PITCHES								TRIAD
0	0	0	4	4	4	0	4	8	12	C	E	G↑	C	C AUG
¯1	0	1	3	4	5	0	3	7	12	C	D↑	G	C	C MIN
¯1	1	0	3	5	4	0	3	8	12	C	D↑	G↑	C	A↓MAJ
0	¯1	1	4	3	5	0	4	7	12	C	E	G	C	C MAJ
0	1	¯1	4	5	3	0	4	9	12	C	E	A	C	A MIN
1	¯1	0	5	3	4	0	5	8	12	C	F	G↑	C	F MIN
1	0	¯1	5	4	3	0	5	9	12	C	F	A	C	F MAJ
¯1	¯1	2	3	3	6	0	3	6	12	C	D↑	F↑	C	C DIM
¯1	2	¯1	3	6	3	0	3	9	12	C	D↑	A	C	A DIM
2	¯1	¯1	6	3	3	0	6	9	12	C	F↑	A	C	F↑DIM
¯2	1	1	2	5	5	0	2	7	12	C	D	G	C	
1	¯2	1	5	2	5	0	5	7	12	C	F	G	C	
1	1	¯2	5	5	2	0	5	10	12	C	F	A↑	C	
¯2	0	2	2	4	6	0	2	6	12	C	D	F↑	C	
¯2	2	0	2	6	4	0	2	8	12	C	D	G↑	C	
0	¯2	2	4	2	6	0	4	6	12	C	E	F↑	C	
0	2	¯2	4	6	2	0	4	10	12	C	E	A↑	C	
2	¯2	0	6	2	4	0	6	8	12	C	F↑	G↑	C	
2	0	¯2	6	4	2	0	6	10	12	C	F↑	A↑	C	
¯2	¯1	3	2	3	7	0	2	5	12	C	D	F	C	
¯2	3	¯1	2	7	3	0	2	9	12	C	D	A	C	
¯1	¯2	3	3	2	7	0	3	5	12	C	D↑	F	C	
¯1	3	¯2	3	7	2	0	3	10	12	C	D↑	A↑	C	
3	¯2	¯1	7	2	3	0	7	9	12	C	G	A	C	
3	¯1	¯2	7	3	2	0	7	10	12	C	G	A↑	C	
¯2	¯2	4	2	2	8	0	2	4	12	C	D	E	C	
¯2	4	¯2	2	8	2	0	2	10	12	C	D	A↑	C	
4	¯2	¯2	8	2	2	0	8	10	12	C	G↑	A↑	C	
¯3	1	2	1	5	6	0	1	6	12	C	C↑	F↑	C	
¯3	2	1	1	6	5	0	1	7	12	C	C↑	G	C	
1	¯3	2	5	1	6	0	5	6	12	C	F	F↑	C	
1	2	¯3	5	6	1	0	5	11	12	C	F	B	C	

2	¯3	1	6	1	5	0	6	7	12	C	F↑	G	C
2	1	¯3	6	5	1	0	6	11	12	C	F↑	B	C
¯3	0	3	1	4	7	0	1	5	12	C	C↑	F	C
¯3	3	0	1	7	4	0	1	8	12	C	C↑	G↑	C
0	¯3	3	4	1	7	0	4	5	12	C	E	F	C
0	3	¯3	4	7	1	0	4	11	12	C	E	B	C
3	¯3	0	7	1	4	0	7	8	12	C	G	G↑	C
3	0	¯3	7	4	1	0	7	11	12	C	G	B	C
¯3	¯1	4	1	3	8	0	1	4	12	C	C↑	E	C
¯3	4	¯1	1	8	3	0	1	9	12	C	C↑	A	C
¯1	¯3	4	3	1	8	0	3	4	12	C	D↑	E	C
¯1	4	¯3	3	8	1	0	3	11	12	C	D↑	B	C
4	¯3	¯1	8	1	3	0	8	9	12	C	G↑	A	C
4	¯1	¯3	8	3	1	0	8	11	12	C	G↑	B	C
¯3	¯2	5	1	2	9	0	1	3	12	C	C↑	D↑	C
¯3	5	¯2	1	9	2	0	1	10	12	C	C↑	A↑	C
¯2	¯3	5	2	1	9	0	2	3	12	C	D	D↑	C
¯2	5	¯3	2	9	1	0	2	11	12	C	D	B	C
5	¯3	¯2	9	1	2	0	9	10	12	C	A	A↑	C
5	¯2	¯3	9	2	1	0	9	11	12	C	A	B	C
¯3	¯3	6	1	1	10	0	1	2	12	C	C↑	D	C
¯3	6	¯3	1	10	1	0	1	11	12	C	C↑	B	C
6	¯3	¯3	10	1	1	0	10	11	12	C	A↑	B	C

Table 11-2. Structures Formed through Distortions of Fourfold Symmetry.

INTERVALS				PITCHES										TETRAD
3	3	3	3	0	3	6	9	12	C	D↑	F↑	A	C	C ∘7
2	3	3	4	0	2	5	8	12	C	D	F	G↑	C	D -MI7
2	3	4	3	0	2	5	9	12	C	D	F	A	C	D MI7
2	4	3	3	0	2	6	9	12	C	D	F↑	A	C	D 7
3	2	3	4	0	3	5	8	12	C	D↑	F	G↑	C	F MI7
3	2	4	3	0	3	5	9	12	C	D↑	F	A	C	F 7
3	3	2	4	0	3	6	8	12	C	D↑	F↑	G↑	C	A↓ 7
3	3	4	2	0	3	6	10	12	C	D↑	F↑	A↑	C	C MI7
3	4	2	3	0	3	7	9	12	C	D↑	G	A	C	A -MI7
3	4	3	2	0	3	7	10	12	C	D↑	G	A↑	C	C MI7
4	2	3	3	0	4	6	9	12	C	E	F↑	A	C	F↑-MI7
4	3	2	3	0	4	7	9	12	C	E	G	A	C	A MI7
4	3	3	2	0	4	7	10	12	C	E	G	A↑	C	C 7
2	2	4	4	0	2	4	8	12	C	D	E	G↑	C	
2	4	2	4	0	2	6	8	12	C	D	F↑	G↑	C	
2	4	4	2	0	2	6	10	12	C	D	F↑	A↑	C	
4	2	2	4	0	4	6	8	12	C	E	F↑	G↑	C	
4	2	4	2	0	4	6	10	12	C	E	F↑	A↑	C	

4	4	2	2	0	4	8	10	12	C	E	G↑	A↑	C		
2	2	3	5	0	2	4	7	12	C	D	E	G	C		
2	2	5	3	0	2	4	9	12	C	D	E	A	C		
2	3	2	5	0	2	5	7	12	C	D	F	G	C		
2	3	5	2	0	2	5	10	12	C	D	F	A↑	C		
2	5	2	3	0	2	7	9	12	C	D	G	A	C		
2	5	3	2	0	2	7	10	12	C	D	G	A↑	C		
3	2	2	5	0	3	5	7	12	C	D↑	F	G	C		
3	2	5	2	0	3	5	10	12	C	D↑	F	A↑	C		
3	5	2	2	0	3	8	10	12	C	D↑	G↑	A↑	C		
5	2	2	3	0	5	7	9	12	C	F	G	A	C		
5	2	3	2	0	5	7	10	12	C	F	G	A↑	C		
5	3	2	2	0	5	8	10	12	C	F	G↑	A↑	C		
2	2	2	6	0	2	4	6	12	C	D	E	F↑	C		
2	2	6	2	0	2	4	10	12	C	D	E	A↑	C		
2	6	2	2	0	2	8	10	12	C	D	G↑	A↑	C		
6	2	2	2	0	6	8	10	12	C	F↑	G↑	A↑	C		
1	3	4	4	0	1	4	8	12	C	C↑	E	G↑	C	C↑	MI
1	4	3	4	0	1	5	8	12	C	C↑	F	G↑	C	C↑	MA
1	4	4	3	0	1	5	9	12	C	C↑	F	A	C	C↑	+
3	1	4	4	0	3	4	8	12	C	D↑	E	G↑	C	E	+
3	4	1	4	0	3	7	8	12	C	D↑	G	G↑	C	A↓	MA
3	4	4	1	0	3	7	11	12	C	D↑	G	B	C	C	MI
4	1	3	4	0	4	5	8	12	C	E	F	G↑	C	F	MI
4	1	4	3	0	4	5	9	12	C	E	F	A	C	F	MA
4	3	1	4	0	4	7	8	12	C	E	G	G↑	C	A↓	+
4	3	4	1	0	4	7	11	12	C	E	G	B	C	C	MA
4	4	1	3	0	4	8	9	12	C	E	G↑	A	C	A	MI
4	4	3	1	0	4	8	11	12	C	E	G↑	B	C	C	+
1	3	3	5	0	1	4	7	12	C	C↑	E	G	C		
1	3	5	3	0	1	4	9	12	C	C↑	E	A	C		
1	5	3	3	0	1	6	9	12	C	C↑	F↑	A	C		
3	1	3	5	0	3	4	7	12	C	D↑	E	G	C		
3	1	5	3	0	3	4	9	12	C	D↑	E	A	C		
3	3	1	5	0	3	6	7	12	C	D↑	F↑	G	C		
3	3	5	1	0	3	6	11	12	C	D↑	F↑	B	C		
3	5	1	3	0	3	8	9	12	C	D↑	G↑	A	C		
3	5	3	1	0	3	8	11	12	C	D↑	G↑	B	C		
5	1	3	3	0	5	6	9	12	C	F	F↑	A	C		
5	3	1	3	0	5	8	9	12	C	F	G↑	A	C		
5	3	3	1	0	5	8	11	12	C	F	G↑	B	C		
1	2	4	5	0	1	3	7	12	C	C↑	D↑	G	C		
1	2	5	4	0	1	3	8	12	C	C↑	D↑	G↑	C		
1	4	2	5	0	1	5	7	12	C	C↑	F	G	C		
1	4	5	2	0	1	5	10	12	C	C↑	F	A↑	C		
1	5	2	4	0	1	6	8	12	C	C↑	F↑	G↑	C		
1	5	4	2	0	1	6	10	12	C	C↑	F↑	A↑	C		
2	1	4	5	0	2	3	7	12	C	D	D↑	G	C		

2	1	5	4	0	2	3	8	12	C	D	D↑	G↑	C
2	4	1	5	0	2	6	7	12	C	D	F↑	G	C
2	4	5	1	0	2	6	11	12	C	D	F↑	B	C
2	5	1	4	0	2	7	8	12	C	D	G	G↑	C
2	5	4	1	0	2	7	11	12	C	D	G	B	C
4	1	2	5	0	4	5	7	12	C	E	F	G	C
4	1	5	2	0	4	5	10	12	C	E	F	A↑	C
4	2	1	5	0	4	6	7	12	C	E	F↑	G	C
4	2	5	1	0	4	6	11	12	C	E	F↑	B	C
4	5	1	2	0	4	9	10	12	C	E	A	A↑	C
4	5	2	1	0	4	9	11	12	C	E	A	B	C
5	1	2	4	0	5	6	8	12	C	F	F↑	G↑	C
5	1	4	2	0	5	6	10	12	C	F	F↑	A↑	C
5	2	1	4	0	5	7	8	12	C	F	G	G↑	C
5	2	4	1	0	5	7	11	12	C	F	G	B	C
5	4	1	2	0	5	9	10	12	C	F	A	A↑	C
5	4	2	1	0	5	9	11	12	C	F	A	B	C

1	2	3	6	0	1	3	6	12	C	C↑	D↑	F↑	C
1	2	6	3	0	1	3	9	12	C	C↑	D↑	A	C
1	3	2	6	0	1	4	6	12	C	C↑	E	F↑	C
1	3	6	2	0	1	4	10	12	C	C↑	E	A↑	C
1	6	2	3	0	1	7	9	12	C	C↑	G	A	C
1	6	3	2	0	1	7	10	12	C	C↑	G	A↑	C
2	1	3	6	0	2	3	6	12	C	D	D↑	F↑	C
2	1	6	3	0	2	3	9	12	C	D	D↑	A	C
2	3	1	6	0	2	5	6	12	C	D	F	F↑	C
2	3	6	1	0	2	5	11	12	C	D	F	B	C
2	6	1	3	0	2	8	9	12	C	D	G↑	A	C
2	6	3	1	0	2	8	11	12	C	D	G↑	B	C
3	1	2	6	0	3	4	6	12	C	D↑	E	F↑	C
3	1	6	2	0	3	4	10	12	C	D↑	E	A↑	C
3	2	1	6	0	3	5	6	12	C	D↑	F	F↑	C
3	2	6	1	0	3	5	11	12	C	D↑	F	B	C
3	6	1	2	0	3	9	10	12	C	D↑	A	A↑	C
3	6	2	1	0	3	9	11	12	C	D↑	A	B	C
6	1	2	3	0	6	7	9	12	C	F↑	G	A	C
6	1	3	2	0	6	7	10	12	C	F↑	G	A↑	C
6	2	1	3	0	6	8	9	12	C	F↑	G↑	A	C
6	2	3	1	0	6	8	11	12	C	F↑	G↑	B	C
6	3	1	2	0	6	9	10	12	C	F↑	A	A↑	C
6	3	2	1	0	6	9	11	12	C	F↑	A	B	C

1	2	2	7	0	1	3	5	12	C	C↑	D↑	F	C
1	2	7	2	0	1	3	10	12	C	C↑	D↑	A↑	C
1	7	2	2	0	1	8	10	12	C	C↑	G↑	A↑	C
2	1	2	7	0	2	3	5	12	C	D	D↑	F	C
2	1	7	2	0	2	3	10	12	C	D	D↑	A↑	C
2	2	1	7	0	2	4	5	12	C	D	E	F	C
2	2	7	1	0	2	4	11	12	C	D	E	B	C
2	7	1	2	0	2	9	10	12	C	D	A	A↑	C
2	7	2	1	0	2	9	11	12	C	D	A	B	C
7	1	2	2	0	7	8	10	12	C	G	G↑	A↑	C
7	2	1	2	0	7	9	10	12	C	G	A	A↑	C

7	2	2	1	0	7	9	11	12	C	G	A	B	C
1	1	5	5	0	1	2	7	12	C	C↑	D	G	C
1	5	1	5	0	1	6	7	12	C	C↑	F↑	G	C
1	5	5	1	0	1	6	11	12	C	C↑	F↑	B	C
5	1	1	5	0	5	6	7	12	C	F	F↑	G	C
5	1	5	1	0	5	6	11	12	C	F	F↑	B	C
5	5	1	1	0	5	10	11	12	C	F	A↑	B	C
1	1	4	6	0	1	2	6	12	C	C↑	D	F↑	C
1	1	6	4	0	1	2	8	12	C	C↑	D	G↑	C
1	4	1	6	0	1	5	6	12	C	C↑	F	F↑	C
1	4	6	1	0	1	5	11	12	C	C↑	F	B	C
1	6	1	4	0	1	7	8	12	C	C↑	G	G↑	C
1	6	4	1	0	1	7	11	12	C	C↑	G	B	C
4	1	1	6	0	4	5	6	12	C	E	F	F↑	C
4	1	6	1	0	4	5	11	12	C	E	F	B	C
4	6	1	1	0	4	10	11	12	C	E	A↑	B	C
6	1	1	4	0	6	7	8	12	C	F↑	G	G↑	C
6	1	4	1	0	6	7	11	12	C	F↑	G	B	C
6	4	1	1	0	6	10	11	12	C	F↑	A↑	B	C
1	1	3	7	0	1	2	5	12	C	C↑	D	F	C
1	1	7	3	0	1	2	9	12	C	C↑	D	A	C
1	3	1	7	0	1	4	5	12	C	C↑	E	F	C
1	3	7	1	0	1	4	11	12	C	C↑	E	B	C
1	7	1	3	0	1	8	9	12	C	C↑	G↑	A	C
1	7	3	1	0	1	8	11	12	C	C↑	G↑	B	C
3	1	1	7	0	3	4	5	12	C	D↑	E	F	C
3	1	7	1	0	3	4	11	12	C	D↑	E	B	C
3	7	1	1	0	3	10	11	12	C	D↑	A↑	B	C
7	1	1	3	0	7	8	9	12	C	G	G↑	A	C
7	1	3	1	0	7	8	11	12	C	G	G↑	B	C
7	3	1	1	0	7	10	11	12	C	G	A↑	B	C
1	1	2	8	0	1	2	4	12	C	C↑	D	E	C
1	1	8	2	0	1	2	10	12	C	C↑	D	A↑	C
1	2	1	8	0	1	3	4	12	C	C↑	D↑	E	C
1	2	8	1	0	1	3	11	12	C	C↑	D↑	B	C
1	8	1	2	0	1	9	10	12	C	C↑	A	A↑	C
1	8	2	1	0	1	9	11	12	C	C↑	A	B	C
2	1	1	8	0	2	3	4	12	C	D	D↑	E	C
2	1	8	1	0	2	3	11	12	C	D	D↑	B	C
2	8	1	1	0	2	10	11	12	C	D	A↑	B	C
8	1	1	2	0	8	9	10	12	C	G↑	A	A↑	C
8	1	2	1	0	8	9	11	12	C	G↑	A	B	C
8	2	1	1	0	8	10	11	12	C	G↑	A↑	B	C
1	1	1	9	0	1	2	3	12	C	C↑	D	D↑	C
1	1	9	1	0	1	2	11	12	C	C↑	D	B	C
1	9	1	1	0	1	10	11	12	C	C↑	A↑	B	C
9	1	1	1	0	9	10	11	12	C	A	A↑	B	C

Table 11-3. Seven-Note Structures Containing only Major and Minor Thirds, Sorted and Grouped by Tension.

36 SIGMA 7'S

3 3 4 4 3 3	C	D↑	F↑	A↑	D	F	G↑	232
3 4 3 4 3 4	C	D↑	G	A↑	D	F	A	232
3 4 4 3 3 4	C	D↑	G	B	D	F	A	232
4 3 3 4 4 3	C	E	G	A↑	D	F↑	A	232
4 3 4 3 4 3	C	E	G	B	D	F↑	A	232
4 4 3 3 4 4	C	E	G↑	B	D	F↑	A↑	233
3 3 4 4 3 4	C	D↑	F↑	A↑	D	F	A	321
3 4 3 4 4 3	C	D↑	G	A↑	D	F↑	A	321
3 4 4 3 4 3	C	D↑	G	B	D	F↑	A	321
4 3 4 4 3 3	C	E	G	B	D↑	F↑	A	321
4 3 4 3 4 4	C	E	G	B	D	F↑	A↑	322
4 4 3 4 3 4	C	E	G↑	B	D↑	F↑	A↑	322
3 4 4 3 4 4	C	D↑	G	B	D	F↑	A↑	411
4 3 4 4 3 4	C	E	G	B	D↑	F↑	A↑	411
4 4 3 4 4 3	C	E	G↑	B	D↑	G	A↑	411
3 3 4 3 4 3	C	D↑	F↑	A↑	C↑	F	G↑	1123
3 4 3 3 4 4	C	D↑	G	A↑	C↑	F	A	1123
3 4 3 4 3 3	C	D↑	G	A↑	D	F	G↑	1123
4 3 3 4 3 4	C	E	G	A↑	D	F	A	1123
4 3 4 3 3 4	C	E	G	B	D	F	A	1123
4 4 3 3 4 3	C	E	G↑	B	D	F↑	A	1123
3 3 4 3 4 4	C	D↑	F↑	A↑	C↑	F	A	1212
4 4 3 4 3 3	C	E	G↑	B	D↑	F↑	A	1212
3 3 3 4 4 3	C	D↑	F↑	A	C↑	F	G↑	1221
3 4 4 3 3 3	C	D↑	G	B	D	F	G↑	1221
3 3 4 3 3 4	C	D↑	F↑	A↑	C↑	E	G↑	2014
3 4 3 3 4 3	C	D↑	G	A↑	C↑	F	G↑	2014
4 3 3 4 3 3	C	E	G	A↑	D	F	G↑	2014
3 3 3 4 3 4	C	D↑	F↑	A	C↑	E	G↑	2112
4 3 3 3 4 4	C	E	G	A↑	C↑	F	A	2112
4 3 4 3 3 3	C	E	G	B	D	F	G↑	2112
4 4 3 3 3 4	C	E	G↑	B	D	F	A	2112
3 3 3 4 3 3	C	D↑	F↑	A	C↑	E	G	3003
3 3 4 3 3 3	C	D↑	F↑	A↑	C↑	E	G	3003
3 4 3 3 3 4	C	D↑	G	A↑	C↑	F	G↑	3003
4 3 3 3 4 3	C	E	G	A↑	C↑	F	G↑	3003

Table 11-4. Seven-Note Structures Suitable for the MT Idiom, Not Restricted in Interval Content.

TNSN	S I G M A							CLASS
232	3 3 4 4 3 3	C	D↑	F↑	A↑	D	F	G↑	-MI7
321	3 3 4 4 3 4	C	D↑	F↑	A↑	D	F	A	-MI7
330	3 3 5 3 3 3	C	D↑	F↑	B	D	F	G↑	-MA7
321	3 3 5 3 3 4	C	D↑	F↑	B	D	F	A	-MA7
421	3 3 5 3 3 5	C	D↑	F↑	B	D	F	A↑	-MA7
232	3 4 3 4 3 4	C	D↑	G	A↑	D	F	A	MI7
321	3 4 3 4 4 3	C	D↑	G	A↑	D	F↑	A	MI7
232	3 4 4 3 3 4	C	D↑	G	B	D	F	A	MI
322	3 4 4 3 3 5	C	D↑	G	B	D	F	A↑	MI
321	3 4 4 3 4 3	C	D↑	G	B	D	F↑	A	MI
411	3 4 4 3 4 4	C	D↑	G	B	D	F↑	A↑	MI
331	3 5 3 3 3 5	C	D↑	G↑	B	D	F	A↑	+MI
322	3 5 3 3 4 4	C	D↑	G↑	B	D	F↑	A↑	+MI
421	3 5 3 3 5 3	C	D↑	G↑	B	D	G	A↑	+MI
1221	4 3 3 3 5 3	C	E	G	A↑	C↑	F↑	A	LG7
232	4 3 3 4 4 3	C	E	G	A↑	D	F↑	A	LG7
321	4 3 3 5 3 3	C	E	G	A↑	D↑	F↑	A	LG7
232	4 3 4 3 4 3	C	E	G	B	D	F↑	A	MA7
322	4 3 4 3 4 4	C	E	G	B	D	F↑	A↑	MA7
1320	4 3 4 4 3 2	C	E	G	B	D↑	F↑	G↑	MA7
321	4 3 4 4 3 3	C	E	G	B	D↑	F↑	A	MA7
411	4 3 4 4 3 4	C	E	G	B	D↑	F↑	A↑	MA7
2122	4 4 2 3 5 1	C	E	G↑	A↑	C↑	F↑	G	+7
1133	4 4 2 4 4 1	C	E	G↑	A↑	D	F↑	G	+7
1222	4 4 2 5 3 1	C	E	G↑	A↑	D↑	F↑	G	+7
233	4 4 3 3 4 4	C	E	G↑	B	D	F↑	A↑	+MA7
322	4 4 3 3 5 3	C	E	G↑	B	D	G	A↑	+MA7
322	4 4 3 4 3 4	C	E	G↑	B	D↑	F↑	A↑	+MA7
411	4 4 3 4 4 3	C	E	G↑	B	D↑	G	A↑	+MA7
1221	5 2 3 3 3 5	C	F	G	A↑	C↑	E	A	7+3
241	5 2 3 4 2 5	C	F	G	A↑	D	E	A	7+3
1222	5 2 3 5 1 5	C	F	G	A↑	D↑	E	A	7+3
241	5 2 4 3 2 5	C	F	G	B	D	E	A	MA7+3
331	5 3 3 3 5 3	C	F	G↑	B	D	G	A↑	+MA7+3
322	5 3 3 4 4 3	C	F	G↑	B	D↑	G	A↑	+MA7+3
421	5 3 3 5 3 3	C	F	G↑	B	E	G	A↑	+MA7+3

Table 11-5. Structures Consisting of Two Strata of Major Triads, Sorted by Tension.

C	E	G	C	E	G	:	3	0
C	E	G	D↑	G	A↑	:	5	101
C	E	G	G	B	D	:	5	111
C	E	G	D	F↑	A	:	6	131
C	E	G	B	D↑	F↑	:	6	310
C	E	G	A	C↑	E	:	5	1010
C	E	G	F	A	C	:	5	1011
C	E	G	A↑	D	F	:	6	1013
C	E	G	E	G↑	B	:	5	1100
C	E	G	G↑	C	D↑	:	5	1100
C	E	G	F↑	A↑	C↑	:	6	1111
C	E	G	C↑	F	G↑	:	6	3001

Table 11-6. Two Strata of Triads Built in Perfect Fourths.

C	F	A↑	C	F	A↑	:	3	1
C	F	A↑	F	A↑	D↑	:	4	2
C	F	A↑	A↑	D↑	G↑	:	5	3
C	F	A↑	G	C	F	:	4	11
C	F	A↑	D	G	C	:	5	21
C	F	A↑	A	D	G	:	6	122
C	F	A↑	E	A	D	:	6	212
C	F	A↑	D↑	G↑	C↑	:	6	1004
C	F	A↑	B	E	A	:	6	1302
C	F	A↑	G↑	C↑	F↑	:	6	2003
C	F	A↑	F↑	B	E	:	6	2202
C	F	A↑	C↑	F↑	B	:	6	3102

Table 11-7. Structures Containing a Triad of Perfect Fourths in the Lower Stratum, with a Triad of Major and Minor Seconds in the Upper Stratum.

C	F	A↑	D	E	F	:	5	121
C	F	A↑	G	A	A↑	:	5	121
C	F	A↑	A↑	C	C↑	:	4	1001
C	F	A↑	D↑	F	F↑	:	5	1002
C	F	A↑	C	D	D↑	:	5	1012
C	F	A↑	F	G	G↑	:	5	1012
C	F	A↑	G↑	A↑	B	:	5	1102
C	F	A↑	A	B	C	:	5	1211
C	F	A↑	C↑	D↑	E	:	6	2112
C	F	A↑	F↑	G↑	A	:	6	2112
C	F	A↑	E	F↑	G	:	6	2121
C	F	A↑	B	C↑	D	:	6	3121

Table 11-8. The 588 Structures Containing Any of Seven Possible Triads in Two Strata.

C	E	G	C	E	G	:	3	0
C	D↑	G	C	D↑	G	:	3	0
C	E	G↑	C	E	G↑	:	3	0
C	E	G↑	E	G↑	C	:	3	0
C	E	G↑	G↑	C	E	:	3	0
C	D↑	F↑	C	D↑	F↑	:	3	0
C	D↑	F↑	D↑	F↑	A	:	4	0
C	D↑	F↑	F↑	A	C	:	4	0
C	D↑	F↑	A	C	D↑	:	4	0
C	E	G	E	G	A↑	:	4	1
C	D↑	G	D↑	G	A↑	:	4	1
C	D↑	F↑	D↑	F↑	A↑	:	4	1
C	F	A↑	C	F	A↑	:	3	1
C	F	A↑	F	G↑	C	:	4	2
C	F	A↑	F	A↑	D↑	:	4	2
C	F	A↑	G↑	C	D↑	:	5	3
C	F	A↑	A↑	D↑	G↑	:	5	3
C	E	G	A	C	E	:	4	10
C	D↑	G	A	C	D↑	:	4	10
C	D↑	F↑	G↑	C	D↑	:	4	10

C	E	G	D	G	C	:	4	11
C	D↑	G	G	C	F	:	4	11
C	F	A↑	A↑	D	F	:	4	11
C	F	A↑	G	C	F	:	4	11
C	E	G	G	A↑	D	:	5	12
C	D↑	G	C	F	A↑	:	5	12
C	D↑	G	F	A↑	D↑	:	5	12
C	D↑	F↑	A↑	D↑	G↑	:	5	12
C	F	A↑	D↑	G	A↑	:	5	12
C	F	A↑	C	D↑	G	:	5	12
C	F	A↑	D	F	G↑	:	5	12
C	E	G	E	A	D	:	5	21
C	E	G	A	D	G	:	5	21
C	D↑	G	F	A	C	:	5	21
C	F	A↑	G	A↑	D	:	5	21
C	F	A↑	D	G	C	:	5	21
C	E	G↑	D	F↑	A↑	:	6	33
C	E	G↑	F↑	A↑	D	:	6	33
C	E	G↑	A↑	D	F↑	:	6	33
C	E	G	C	D↑	G	:	4	100
C	E	G	E	G	B	:	4	100
C	D↑	G	D↑	G	B	:	4	100
C	D↑	G	G	B	D↑	:	4	100
C	D↑	G	B	D↑	G	:	4	100
C	D↑	G	C	D↑	F↑	:	4	100
C	D↑	G	G	C	F↑	:	4	100
C	E	G↑	C	E	G	:	4	100
C	E	G↑	E	G↑	B	:	4	100
C	E	G↑	G↑	C	D↑	:	4	100
C	D↑	F↑	B	D↑	F↑	:	4	100
C	D↑	F↑	C	F↑	B	:	4	100
C	F	B	F	G↑	C	:	4	100
C	F	B	F	G↑	B	:	4	100
C	F	B	C	F	B	:	3	100
C	F↑	B	B	D↑	F↑	:	4	100
C	F↑	B	C	D↑	F↑	:	4	100
C	F↑	B	C	F↑	B	:	3	100
C	E	G	D↑	G	A↑	:	5	101
C	E	G	E	A↑	D↑	:	5	101
C	D↑	G	D↑	F↑	A↑	:	5	101
C	D↑	F↑	A↑	D↑	A	:	5	101
C	F	A↑	F	A	C	:	4	101
C	F	A↑	F	A↑	E	:	4	101
C	F	B	F	A	C	:	4	101
C	F	B	G↑	C	D↑	:	5	101
C	F	B	G↑	B	D↑	:	5	101
C	F↑	B	D↑	F↑	A	:	5	101
C	F↑	B	F↑	A	C	:	4	101
C	F↑	B	A	C	D↑	:	5	101

C	F↑	B	F↑	B	E	:	4	101
C	F	A↑	C	E	G↑	:	5	102
C	F	A↑	E	G↑	C	:	5	102
C	F	A↑	G↑	C	E	:	5	102
C	F	A↑	A	C	D↑	:	5	102
C	F	A↑	A↑	D↑	A	:	5	102
C	F	B	A	C	D↑	:	5	102
C	F↑	B	A	C	E	:	5	102
C	F↑	B	B	E	A	:	5	102
C	E	G	A	C	D↑	:	5	110
C	E	G	E	A	D↑	:	5	110
C	E	G	G	C	F↑	:	4	110
C	D↑	G	D↑	F↑	A	:	5	110
C	D↑	G	F↑	A	C	:	5	110
C	D↑	G	D	G	C	:	4	110
C	D↑	F↑	D	F↑	A	:	5	110
C	D↑	F↑	F	A	C	:	5	110
C	D↑	F↑	G↑	B	D↑	:	5	110
C	D↑	F↑	D↑	A	D	:	5	110
C	D↑	F↑	F↑	C	F	:	4	110
C	D↑	F↑	A	D↑	G↑	:	5	110
C	F	B	D	F	G↑	:	5	110
C	F	B	G↑	B	D	:	5	110
C	F	B	B	D	F	:	4	110
C	F	B	G	C	F	:	4	110
C	F↑	B	G↑	C	D↑	:	5	110
C	F↑	B	G↑	B	D↑	:	5	110
C	F↑	B	B	D	F↑	:	4	110
C	E	G	G	B	D	:	5	111
C	E	G	B	E	A	:	5	111
C	E	G	A↑	E	A	:	5	111
C	D↑	G	G	A↑	D	:	5	111
C	D↑	G	C	F	B	:	5	111
C	D↑	G	A↑	D↑	A	:	5	111
C	E	G↑	C	D↑	F↑	:	5	111
C	E	G↑	E	G	A↑	:	5	111
C	E	G↑	G↑	B	D	:	5	111
C	E	G↑	D	G	C	:	5	111
C	E	G↑	F↑	B	E	:	5	111
C	E	G↑	A↑	D↑	G↑	:	5	111
C	E	G↑	C	F↑	B	:	5	111
C	E	G↑	E	A↑	D↑		5	111
C	E	G↑	G↑	D	G	:	5	111
C	D↑	F↑	D	F↑	A↑	:	5	111
C	D↑	F↑	F↑	A↑	D	:	5	111
C	D↑	F↑	A↑	D	F↑	:	5	111
C	D↑	F↑	C	F	A↑	:	5	111
C	D↑	F↑	F	A↑	D↑	:	5	111
C	F	A↑	C	E	G	:	5	111
C	F	A↑	D	F	A	:	5	111
C	F	A↑	E	G	A↑	:	5	111

C	F	B	C	D↑	G	:	5	111
C	F	B	D	F	A	:	5	111
C	F	B	D↑	G	B	:	5	111
C	F	B	G	B	D↑	:	5	111
C	F	B	B	D↑	G	:	5	111
C	F↑	B	D	F↑	A	:	5	111
C	F↑	B	E	G↑	B	:	5	111
C	F↑	B	C	E	G↑	:	5	111
C	F↑	B	E	G↑	C	:	5	111
C	F↑	B	G↑	C	E	:	5	111
C	F	A↑	D↑	G↑	D	:	6	113
C	F↑	B	E	A	D	:	6	113
C	E	G	F↑	A	C	:	5	120
C	D↑	G	A	D	G	:	5	120
C	D↑	G	D↑	A	D	:	5	120
C	D↑	F↑	F	G↑	C	:	5	120
C	D↑	F↑	D↑	G↑	D	:	5	120
C	F	B	G	B	D	:	5	120
C	F	B	D	G	C	:	5	120
C	F↑	B	G↑	B	D	:	5	120
C	E	G	D	F↑	A↑	:	6	122
C	E	G	F↑	A↑	D	:	6	122
C	E	G	A↑	D	F↑	:	6	122
C	D↑	G	A↑	D	F	:	6	122
C	E	G↑	D↑	F↑	A↑	:	6	122
C	E	G↑	G	A↑	D	:	6	122
C	E	G↑	B	D	F↑	:	6	122
C	F	A↑	A	D	G	:	6	122
C	F	A↑	G↑	D	G	:	6	122
C	F	B	A	D	G	:	6	122
C	E	G	D	F↑	A	:	6	131
C	D↑	G	D	F	A	:	6	131
C	E	G	D↑	G	B	:	5	200
C	E	G	G	B	D↑	:	5	200
C	E	G	B	D↑	G	:	5	200
C	D↑	G	B	D↑	F↑	:	5	200
C	D↑	G	C	F↑	B	:	5	200
C	E	G↑	C	D↑	G	:	5	200
C	E	G↑	E	G	B	:	5	200
C	E	G↑	G↑	B	D↑	:	5	200
C	F	B	E	G↑	B	:	5	200
C	F	B	C	E	G↑	:	5	200
C	F	B	E	G↑	C	:	5	200
C	F	B	G↑	C	E	:	5	200
C	F	B	F	B	E	:	4	200
C	F↑	B	C	F	B	:	4	200
C	F↑	B	F↑	B	F	:	4	200
C	F↑	B	F↑	C	F	:	4	200

C	E	G	B	E	A↑	:	5	201
C	F	A↑	A	C	E	:	5	201
C	F	A↑	A↑	E	A	:	5	201
C	F	B	A	C	E	:	5	201
C	F	B	C	F	A↑	:	4	201
C	F	B	B	E	A	:	5	201
C	F	B	B	F	A↑	:	4	201
C	F↑	B	F	A	C	:	5	201
C	F↑	B	D↑	F↑	A↑	:	5	201
C	F	A↑	E	A↑	D↑	:	5	202
C	F	B	F	A↑	D↑	:	5	202
C	F	B	A	D↑	G↑	:	6	202
C	F↑	B	E	A	D↑	:	6	202
C	F↑	B	B	E	A↑	:	5	202
C	F	A↑	A	D↑	G↑	:	6	203
C	F	B	A↑	D↑	G↑	:	6	203
C	E	G	C	D↑	F↑	:	5	210
C	E	G	F↑	B	E	:	5	210
C	E	G	C	F↑	B	:	5	210
C	D↑	G	G	B	D	:	5	210
C	D↑	F↑	B	D	F↑	:	5	210
C	D↑	F↑	C	F	B	:	5	210
C	D↑	F↑	F↑	B	F	:	5	210
C	F	B	C	E	G	:	5	210
C	F	B	E	G	B	:	5	210
C	F↑	B	F	G↑	C	:	5	210
C	F↑	B	F	G↑	B	:	5	210
C	F↑	B	B	D	F	:	5	210
C	E	G	D↑	F↑	A↑	:	6	211
C	E	G	A↑	D↑	A	:	6	211
C	D↑	G	D	F↑	A↑	:	6	211
C	D↑	G	F↑	A↑	D	:	6	211
C	D↑	G	A↑	D	F↑	:	6	211
C	D↑	G	F↑	C	F	:	5	211
C	E	G↑	D↑	G	A↑	:	6	211
C	E	G↑	G	B	D	:	6	211
C	E	G↑	B	D↑	F↑	:	6	211
C	E	G↑	D↑	G↑	D	:	5	211
C	E	G↑	G	C	F↑	:	5	211
C	E	G↑	B	E	A↑	:	5	211
C	F	B	A↑	D	F	:	5	211
C	F	B	D↑	G↑	D	:	6	211
C	F	B	A	D	G↑	:	6	211
C	F↑	B	D	F	A	:	6	211
C	F↑	B	D	F↑	A↑	:	5	211
C	F↑	B	F↑	A↑	D	:	5	211
C	F↑	B	A↑	D	F↑	:	5	211
C	F↑	B	D↑	A	D	:	6	211
C	F↑	B	A	D↑	G↑	:	6	211
C	D↑	G	B	F	A↑	:	6	212

C	F	A↑	E	A	D	:	6	212
C	F	A↑	A	D	G↑	:	6	212
C	F	A↑	D↑	A	D	:	6	212
C	F	B	D↑	G	A↑	:	6	212
C	F	B	E	A	D	:	6	212
C	F	B	D↑	A	D	:	6	212
C	F↑	B	A↑	D↑	G↑	:	6	212
C	E	G	D↑	F↑	A	:	6	220
C	D↑	G	D	F↑	A	:	6	220
C	D↑	F↑	D	F	A	:	6	220
C	D↑	F↑	F	G↑	B	:	6	220
C	D↑	F↑	G↑	B	D	:	6	220
C	D↑	F↑	A	D	G↑	:	6	220
C	F	B	G↑	D	G	:	6	220
C	F↑	B	D	F	G↑	:	6	220
C	F↑	B	D↑	G↑	D	:	6	220
C	E	G	B	D	F↑	:	6	221
C	E	G	D↑	A	D	:	6	221
C	D↑	G	B	D	F	:	6	221
C	D↑	F↑	A↑	D	F	:	6	221
C	F	B	G	A↑	D	:	6	221
C	F↑	B	A	D	G↑	:	6	221
C	D↑	F↑	D	F	G↑	:	6	230
C	E	G↑	D↑	G	B	:	6	300
C	E	G↑	G	B	D↑	:	6	300
C	E	G↑	B	D↑	G	:	6	300
C	F	B	F	A↑	E	:	5	301
C	F	B	B	E	A↑	:	5	301
C	F↑	B	C	F	A↑	:	5	301
C	F↑	B	F	B	E	:	5	301
C	F↑	B	B	F	A↑	:	5	301
C	F	A↑	E	A	D↑	:	6	302
C	F	B	E	A	D↑	:	6	302
C	F↑	B	F	A↑	D↑	:	6	302
C	F↑	B	A↑	D↑	A	:	6	302
C	F↑	B	E	A↑	D↑	:	6	302
C	F	B	A↑	D↑	A	:	6	303
C	F↑	B	A↑	E	A	:	6	303
C	E	G	B	D↑	F↑	:	6	310
C	D↑	G	B	D	F↑	:	6	310
C	D↑	G	F↑	B	F	:	6	311
C	D↑	F↑	B	F	A↑	:	6	311
C	F	B	E	G	A↑	:	6	311
C	F↑	B	A↑	D	F	:	6	311
C	D↑	F↑	B	D	F	:	6	320

C	F	B	E	A↑	D↑	:	6	402
C	F	B	A↑	E	A	:	6	402
C	F↑	B	F	A↑	E	:	6	402
C	E	G	C	E	G↑	:	4	1000
C	E	G	E	G↑	C	:	4	1000
C	E	G	G↑	C	E	:	4	1000
C	E	G	C↑	E	G	:	4	1000
C	E	G	C↑	G	C	:	4	1000
C	D↑	G	C	E	G	:	4	1000
C	D↑	G	G↑	C	D↑	:	4	1000
C	E	G↑	C↑	E	G↑	:	4	1000
C	E	G↑	F	G↑	C	:	4	1000
C	E	G↑	A	C	E	:	4	1000
C	D↑	F↑	C	D↑	G	:	4	1000
C	D↑	F↑	G	C	F↑	:	4	1000
C	E	G	G	A↑	C↑	:	5	1001
C	E	G	A↑	C↑	E	:	5	1001
C	E	G	G	C	F	:	4	1001
C	D↑	G	E	G	A↑	:	5	1001
C	D↑	G	C↑	G	C	:	4	1001
C	D↑	G	E	A↑	D↑	:	5	1001
C	D↑	F↑	D↑	G	A↑	:	5	1001
C	D↑	F↑	F↑	A	C↑	:	5	1001
C	D↑	F↑	A	C	E	:	5	1001
C	D↑	F↑	C↑	F↑	C	:	4	1001
C	D↑	F↑	E	A	D↑	:	5	1001
C	F	A↑	A↑	C↑	F	:	4	1001
C	F	A↑	F↑	C	F	:	4	1001
C	E	G	C	F	A↑	:	5	1002
C	E	G	F	A↑	E	:	5	1002
C	D↑	G	G	A↑	C↑	:	5	1002
C	D↑	G	A↑	D↑	G↑	:	5	1002
C	D↑	F↑	F↑	A↑	C↑	:	5	1002
C	D↑	F↑	E	A↑	D↑	:	5	1002
C	F	A↑	C↑	F	G↑	:	5	1002
C	F	A↑	D↑	F↑	A↑	:	5	1002
C	F	A↑	C	D↑	F↑	:	5	1002
C	F	A↑	D↑	G↑	C↑	:	6	1004
C	E	G	A	C↑	E	:	5	1010
C	D↑	G	A	C	E	:	5	1010
C	D↑	G	E	A	D↑	:	5	1010
C	E	G	F	A	C	:	5	1011
C	E	G	G↑	D	G	:	5	1011
C	D↑	G	F	G↑	C	:	5	1011
C	E	G↑	D	F	G↑	:	5	1011
C	E	G↑	F↑	A	C	:	5	1011
C	E	G↑	A↑	C↑	E	:	5	1011
C	E	G↑	C	F	A↑	:	5	1011
C	E	G↑	E	A	D	:	5	1011

C	E	G↑	G↑	C↑	F↑	:	5	1011
C	E	G↑	C↑	F↑	C	:	5	1011
C	E	G↑	F	A↑	E	:	5	1011
C	E	G↑	A	D	G↑	:	5	1011
C	D↑	F↑	C	E	G↑	:	5	1011
C	D↑	F↑	E	G↑	C	:	5	1011
C	D↑	F↑	G↑	C	E	:	5	1011
C	D↑	F↑	D↑	G↑	C↑	:	5	1011
C	D↑	F↑	G↑	C↑	F↑	:	5	1011
C	F	A↑	D	F↑	A↑	:	5	1011
C	F	A↑	F↑	A↑	D	:	5	1011
C	F	A↑	A↑	D	F↑	:	5	1011
C	F	A↑	G	A↑	C↑	:	5	1011
C	F	A↑	C↑	G	C	:	5	1011
C	E	G	A↑	D	F	:	6	1013
C	D↑	G	A↑	C↑	F	:	6	1013
C	E	G	D	F	A	:	6	1022
C	D↑	G	C↑	F	A	:	6	1022
C	D↑	G	F	A	C↑	:	6	1022
C	D↑	G	A	C↑	F	:	6	1022
C	E	G↑	D	F↑	A	:	6	1022
C	E	G↑	F↑	A↑	C↑	:	6	1022
C	E	G↑	A↑	D	F	:	6	1022
C	E	G	C	G↑	B	:	5	1100
C	E	G	G↑	C	D↑	:	5	1100
C	D↑	G	E	G	B	:	5	1100
C	D↑	G	G↑	B	D↑	:	5	1100
C	E	G↑	C↑	E	G	:	5	1100
C	E	G↑	F	G↑	B	:	5	1100
C	E	G↑	A	C	D↑	:	5	1100
C	E	G↑	C	F	B	:	5	1100
C	E	G↑	E	A	D↑	:	5	1100
C	E	G↑	G↑	C↑	G	:	5	1100
C	E	G↑	C↑	G	C	:	5	1100
C	E	G↑	F	B	E	:	5	1100
C	E	G↑	A	D↑	G↑		5	1100
C	D↑	F↑	D↑	G	B	:	5	1100
C	D↑	F↑	G	B	D↑	:	5	1100
C	D↑	F↑	B	D↑	G	:	5	1100
C	F	B	F↑	B	F	:	4	1100
C	F	B	C	F↑	B	:	4	1100
C	F	B	F↑	C	F	:	4	1100
C	F↑	B	C	D↑	G	:	5	1100
C	F↑	B	D↑	G	B	:	5	1100
C	F↑	B	G	B	D↑	:	5	1100
C	F↑	B	B	D↑	G	:	5	1100
C	F↑	B	G	C	F↑	:	4	1100
C	E	G	C	F	B	:	5	1101
C	E	G	F	B	E	:	5	1101
C	D↑	G	C↑	F↑	C	:	5	1101
C	D↑	G	G	C↑	F↑	:	5	1101

C	E	G↑	D↑	G↑	C↑	:	5	1101
C	E	G↑	G	C	F	:	5	1101
C	E	G↑	B	E	A	:	5	1101
C	D↑	F↑	F↑	B	E	:	5	1101
C	F	A↑	C↑	F	A	:	5	1101
C	F	A↑	F	A	C↑	:	5	1101
C	F	A↑	A	C↑	F	:	5	1101
C	F	A↑	F↑	A	C	:	5	1101
C	F	A↑	A↑	C↑	E	:	5	1101
C	F	A↑	C	F	B	:	4	1101
C	F	A↑	B	F	A↑	·	4	1101
C	F	B	B	D↑	F↑	:	5	1101
C	F	B	C	D↑	F↑	·	5	1101
C	F	B	F↑	A	C	:	5	1101
C	F↑	B	C	E	G	:	5	1101
C	F↑	B	E	G	B	:	5	1101

C	E	G	A↑	D↑	G↑	:	6	1102
C	D↑	G	F↑	A↑	C↑	.	6	1102
C	D↑	F↑	C↑	F↑	B		5	1102
C	D↑	F↑	B	E	A	:	6	1102
C	D↑	F↑	A↑	E	A	:	6	1102
C	F	A↑	C↑	E	G↑	:	6	1102
C	F	A↑	D↑	F↑	A	:	6	1102
C	F	A↑	F	G↑	B	:	5	1102
C	F	B	D↑	F↑	A	:	6	1102

C	E	G	F	A↑	D↑	·	6	1103
C	F	A↑	G↑	B	D↑	·	6	1103

C	E	G	C↑	F↑	C	:	5	1110
C	E	G	G	C↑	F↑	:	5	1110
C	D↑	G	D↑	G↑	D	:	5	1110
C	D↑	G	G↑	D	G	:	5	1110
C	D↑	G	A	D↑	G↑	:	5	1110
C	D↑	F↑	D	G	C	:	5	1110
C	F	B	C↑	F	G↑	:	5	1110
C	F	B	B	D	F↑	:	5	1110
C	F↑	B	G	B	D	:	5	1110
C	F↑	B	C↑	F↑	B	:	4	1110
C	F↑	B	D	G	C	:	5	1110
C	F↑	B	C↑	F↑	C	:	4	1110

C	E	G	F↑	A↑	C↑	:	6	1111
C	E	G	G↑	B	D	:	6	1111
C	E	G	D	G	C↑	:	5	1111
C	D↑	G	F↑	A	C↑	:	6	1111
C	D↑	G	F	G↑	B	:	6	1111
C	D↑	G	B	E	A	:	6	1111
C	D↑	G	A↑	E	A	:	6	1111
C	E	G↑	D↑	F↑	A	:	6	1111
C	E	G↑	G	A↑	C↑	:	6	1111
C	F	G↑	B	D	F	:	6	1111
C	E	G↑	D	G↑	C↑	:	5	1111

C	E	G↑	F↑	C	F	:	5	1111
C	E	G↑	A↑	E	A	:	5	1111
C	D↑	F↑	E	G↑	B	:	6	1111
C	D↑	F↑	G	A↑	D	:	6	1111
C	D↑	F↑	C↑	F	A	:	6	1111
C	D↑	F↑	F	A	C↑	:	6	1111
C	D↑	F↑	A	C↑	F	:	6	1111
C	D↑	F↑	G	C	F	:	5	1111
C	D↑	F↑	A	D	G	:	6	1111
C	F	A↑	D	F↑	A	:	6	1111
C	F	A↑	C↑	E	G	:	6	1111
C	F	A↑	B	D	F	:	5	1111
C	F	A↑	G	C	F↑	:	5	1111
C	F	B	D	F↑	A	:	6	1111
C	F	B	C↑	F	A	:	5	1111
C	F	B	F	A	C↑	:	5	1111
C	F	B	A	C↑	F	:	5	1111
C	F↑	B	F↑	A	C↑	:	5	1111

C	E	G	B	D	F	:	6	1112
C	D↑	G	F	A↑	E	:	6	1112
C	E	G↑	C↑	F↑	B	:	6	1112
C	E	G↑	F	A↑	D↑	:	6	1112
C	E	G↑	A	D	G	:	6	1112
C	D↑	F↑	A↑	C↑	F	:	6	1112
C	D↑	F↑	E	A	D	:	6	1112
C	F	A↑	D↑	G	B	:	6	1112
C	F	A↑	G	B	D↑	:	6	1112
C	F	A↑	B	D↑	G	:	6	1112
C	F	A↑	G↑	B	D	:	6	1112
C	F	A↑	G↑	C↑	G	:	6	1112
C	F	A↑	D	G↑	C↑	:	6	1112
C	F	B	D↑	G↑	C↑	:	6	1112
C	F↑	B	A	C↑	E	:	6	1112
C	F↑	B	A	D	G	:	6	1112

C	E	G	F↑	A	C↑	:	6	1120
C	F	B	C↑	G	C	:	5	1120
C	F↑	B	G↑	C↑	F↑	:	5	1120

C	E	G	A	D	G↑	:	6	1121
C	D↑	G	D	F	G↑	:	6	1121
C	D↑	G	E	A	D	:	6	1121
C	D↑	F↑	C↑	F	G↑	:	6	1121
C	F	A↑	G	B	D	:	6	1121
C	F	A↑	D	G	C↑	:	6	1121
C	F↑	B	C↑	E	G↑	:	6	1121
C	F↑	B	D↑	G↑	C↑	:	6	1121

C	E	G	G↑	B	D↑	:	6	1200
C	D↑	G	F↑	B	E	:	6	1201
C	D↑	G	B	E	A↑	:	6	1201
C	F	A↑	A	C↑	E	:	6	1201
C	F	A↑	B	E	A↑	:	5	1201

C	F	A↑	F	B	E	:	5	1201
C	F	B	F↑	B	E	:	5	1201
C	F↑	B	D↑	G	A↑	:	6	1201
C	F↑	B	G	C	F	:	5	1201
C	E	G	B	F	A↑	:	6	1202
C	D↑	G	C↑	F↑	B	:	6	1202
C	D↑	F↑	B	E	A↑	:	6	1202
C	F	A↑	E	G↑	B	:	6	1202
C	F	B	D↑	F↑	A↑	:	6	1202
C	F↑	B	E	G	A↑	:	6	1202
C	E	G	A	D↑	G↑	:	6	1210
C	D↑	G	G↑	B	D	:	6	1210
C	D↑	F↑	G	B	D	:	6	1210
C	F	B	C↑	E	G↑	:	6	1210
C	F	B	G	C	F↑	:	5	1210
C	E	G	C↑	F↑	B	:	6	1211
C	E	G	D↑	G↑	D	:	6	1211
C	E	G	F↑	C	F	:	5	1211
C	D↑	G	D	G	C↑	:	5	1211
C	D↑	G	F	B	E	:	6	1211
C	E	G↑	D	G	C↑	:	6	1211
C	E	G↑	F↑	B	F	:	6	1211
C	E	G↑	A↑	D↑	A	:	6	1211
C	E	G↑	D↑	A	D	:	6	1211
C	E	G↑	G	C↑	F↑	:	6	1211
C	E	G↑	B	F	A↑	:	6	1211
C	F	A↑	E	G	B	:	6	1211
C	F	B	A	C↑	E	:	6	1211
C	F	B	A↑	C↑	F	:	5	1211
C	F	B	D	F↑	A↑	:	6	1211
C	F	B	F↑	A↑	D	:	6	1211
C	F	B	A↑	D	F↑	:	6	1211
C	F↑	B	F↑	A↑	C↑	:	5	1211
C	F↑	B	G	A↑	D	:	6	1211
C	F↑	B	C↑	F	A	:	6	1211
C	F↑	B	F	A	C↑	:	6	1211
C	F↑	B	A	C↑	F	:	6	1211
C	D↑	F↑	F	A↑	E	:	6	1212
C	F↑	B	A↑	C↑	E	:	6	1212
C	D↑	G	A	D	G↑	:	6	1220
C	D↑	F↑	G↑	D	G	:	6	1220
C	F	B	C↑	E	G	:	6	1220
C	F	B	G↑	C↑	G	:	6	1220
C	F	B	D	G↑	C↑	:	6	1220
C	F↑	B	C↑	F	G↑	:	6	1220
C	F↑	B	G↑	D	G	:	6	1220
C	D↑	F↑	D	G↑	C↑	:	6	1221
C	F	B	G	A↑	C↑	:	6	1221

C	F	B	D	G	C↑	:	6 1230
C	F↑	B	D	G↑	C↑	:	6 1230

C	F	A↑	B	E	A	:	6 1302
C	E	G	F↑	B	F	:	6 1311
C	D↑	F↑	F	B	E	:	6 1311
C	F	B	A↑	C↑	E	:	6 1311
C	F↑	B	A↑	C↑	F	:	6 1311

C	E	G	C↑	E	G↑	:	5 2000
C	E	G	G↑	C↑	G	:	5 2000
C	D↑	G	C	E	G↑	:	5 2000
C	D↑	G	E	G↑	C	:	5 2000
C	D↑	G	G↑	C	E	:	5 2000
C	E	G↑	C↑	F	G↑	:	5 2000
C	E	G↑	F	A	C	:	5 2000
C	E	G↑	A	C↑	E	:	5 2000

C	E	G	F	G↑	C	:	5 2001
C	D↑	G	C↑	E	G	:	5 2001
C	D↑	G	D↑	G↑	C↑	:	5 2001
C	D↑	G	G↑	C↑	G	:	5 2001
C	D↑	F↑	C	E	G	:	5 2001
C	D↑	F↑	C↑	G	C	:	5 2001
C	D↑	F↑	G	C↑	F↑	:	5 2001
C	F	A↑	F↑	A↑	C↑	:	5 2001
C	F	A↑	C↑	F↑	C	:	5 2001

C	E	G	A↑	C↑	F	:	6 2002
C	D↑	G	A↑	C↑	E	:	6 2002
C	D↑	F↑	A	C↑	E	:	6 2002
C	D↑	F↑	E	G	A↑	:	6 2002
C	D↑	F↑	G	A↑	C↑	:	6 2002
C	D↑	F↑	A↑	C↑	E	:	6 2003
C	F	A↑	G↑	C↑	F↑	:	6 2003

C	E	G	C↑	F	A	:	6 2011
C	E	G	F	A	C↑	:	6 2011
C	E	G	A	C↑	F	:	6 2011
C	D↑	G	A	C↑	E	:	6 2011
C	E	G↑	D	F	A	:	6 2011
C	E	G↑	F↑	A	C↑	:	6 2011
C	E	G↑	A↑	C↑	F	:	6 2011

C	E	G	D	F	G↑	:	6 2012
C	D↑	G	C↑	F	G↑	:	6 2012
C	D↑	F↑	C↑	E	G↑	:	6 2012

C	D↑	G	E	G↑	B	:	6 2100

C	E	G	F	G↑	B	:	6 2101
C	E	G	D↑	G↑	C↑	:	6 2101
C	D↑	F↑	E	G	B	:	6 2101
C	F	A↑	F↑	A	C↑	:	6 2101

C	F	A↑	F↑	B	F	:	5	2101
C	F	A↑	C	F↑	B	:	5	2101
C	D↑	G	G↑	C↑	F↑	:	6	2102
C	F	A↑	B	D↑	F↑	:	6	2102
C	F	B	C↑	F↑	B	:	5	2110
C	F	B	C↑	F↑	C	:	5	2110
C	F↑	B	C↑	G	C	:	5	2110
C	F↑	B	G	C↑	F↑	:	5	2110
C	E	G	G↑	C↑	F↑	:	6	2111
C	E	G	D	G↑	C↑	:	6	2111
C	D↑	F↑	G↑	C↑	G	:	6	2111
C	F	A↑	B	D	F↑	:	6	2111
C	F	A↑	G	C↑	F↑	:	6	2111
C	F	B	F↑	A	C↑	:	6	2111
C	F	B	G↑	C↑	F↑	:	6	2111
C	F↑	B	C↑	E	G	:	6	2111
C	F	A↑	F↑	B	E	:	6	2202
C	D↑	G	D	G↑	C↑	:	6	2211
C	D↑	F↑	D	G	C↑	:	6	2211
C	F	B	F↑	A↑	C↑	:	6	2211
C	F↑	B	G	A↑	C↑	:	6	2211
C	F	B	G	C↑	F↑	:	6	2220
C	F↑	B	D	G	C↑	:	6	2220
C	F↑	B	G↑	C↑	G	:	6	2220
C	E	G↑	C↑	F	A	:	6	3000
C	E	G↑	F	A	C↑	:	6	3000
C	E	G↑	A	C↑	F	:	6	3000
C	E	G	C↑	F	G↑	:	6	3001
C	D↑	G	C↑	E	G↑	:	6	3001
C	D↑	F↑	C↑	E	G	:	6	3002
C	F	A↑	C↑	F↑	B	:	6	3102
C	F	B	C↑	F↑	C		5	3210
C	F↑	B	C↑	G	C		5	3210

12 Combining Harmony and Melody

The "simultaneous" composition of harmony and melody can, in theory, be carried out in two different ways; the word appears in quotes because both methods consist of two *sequential* steps. The names given the two procedures couldn't describe those steps more clearly—harmonization of melody and melodization of harmony. But the composition of harmony and melody is, in practice, carried out through an unpredictable mixture of these methods that may seem "simultaneous" to the composer. He might begin with a melodic motif, find that it cries out for a particular harmonization, and then find this harmony suggests a short sequence of chords to follow which, in turn, brings to mind its own melodization. The process continues in this almost self-directing way, though of course, control and responsibility are the composer's.

Our ultimate goal will be to create algorithms that let the composer work in just this way, but that requires a thorough understanding of both techniques. Melodization of harmony is by far the easier method. In fact, before I had access to APL, it was the only technique I programmed. I was unable to collect enough energy to attack the problem of harmonization of melody until the more powerful language was available. When I thought about reprogramming my melodization techniques in APL, I realized there was practically nothing to program. The original effort went mostly into the creation of an interactive environment that could handle vectors and matrices! As a result, I will briefly describe melodization techniques first, and then go into harmonization in far more detail.

MELODIZATION OF HARMONY

This technique is used instinctively by amateurs and as a weapon against instinct by pros. The amateur relies on the enormous collection of musical clichés that pervade the MT idiom. Each cliché consists of a sequence of a few chords and any number of stylized melodizations. After the amateur selects his first few melodic tones, he immediately "hears" one of these clichés fall into place and it then guides him through the rest of the
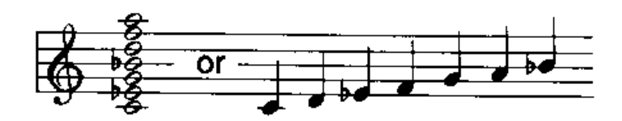


Fig. 12-1. The sixth entry in Tables 11-4 (chord) and 12-1 (scale).

first motif. If a "connecting" cliché doesn't lead the way from there, his own approach to development will help him begin the next motif with one or two notes, and again he will "lock in" to a cliché and carry on.

Professional composers "hear" these same clichés, but do their best to avoid them. One way to do this is to compose an unusual chord progression and then find a melody that fits it. Without the new progression, habitual forces can restrict one's melodic choices to the same old hackneyed patterns.

Having no instinct, the computer must depend on our programs to compose sequences of chords and to fit melodies to given chords. We'll reserve the problem of composing chordal sequences for the following chapters. Here we'll assume that, at any given time, there is a given chord that needs to be melodized. The seven-note structures described in the previous chapter suggest a convenient basis for method here because their complete tonality can be spelled out as scales for melody as well as for harmony. Table 12-1 at the end of the chapter shows the same structures that appear in Table 11-4, but compressed into scales.

Now suppose the harmony that needs melodization is the sixth one in our list, a C-minor-seventh structure (Fig. 12-1). In APL, the complete harmonic structure can be represented by the vector

```
HRM ← 0  3  7  10  2  5  9
```

and the scale by

```
SCL ← 0  2  3  5  7  9  10.
```

A chord might now be chosen using

```
CHD ← HRM[0 1 3]
```

and a melodic fragment

```
M ← SCL[4 3 4 5 1 2]
```

all of which combine to produce Fig. 12-2.

At first, the problem of finding a melody to fit a given chord seems only to require the selection of numbers (structure indices) from zero to six. But from all that has been said about sensibility, it should be clear that the individual snippets of melody that go with each chord must also fit together to satisfy the requirements of structure in the larger sense. This means that two critical problems must be solved in a melodization algorithm. First, though the numbers that generate the melody are now indices to harmonic structures, the melodic motifs must still show structural relatedness. And second, the pieces of melody that fit the individual chords must link together without awkwardness.

A Geometric Approach to Melodization

A method that could take us too far afield uses computer graphics and what Schillinger called *melodic trajectories* and the *axes of melody* in pitch/time space. Briefly, this involves the construction of "curves" that will guide the melody. Although continuous (though not necessarily smooth) curves are used, our psychological need for quantization in pitch and time requires that the curves be "sampled" to extract just those points whose time coordinates correspond to the rhythmic attacks of the melody. Each sampling then provides a point whose pitch coordinate needs to be adjusted
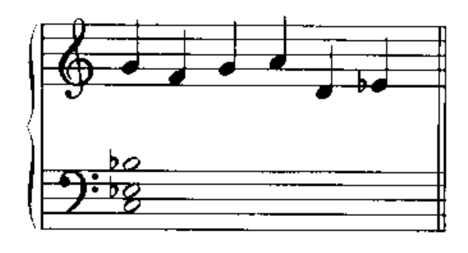


Fig. 12-2. A chord and melodic fragment selected from a seven-note structure.

to the nearest tone in the harmonic structure that is being melodized at that point in time.
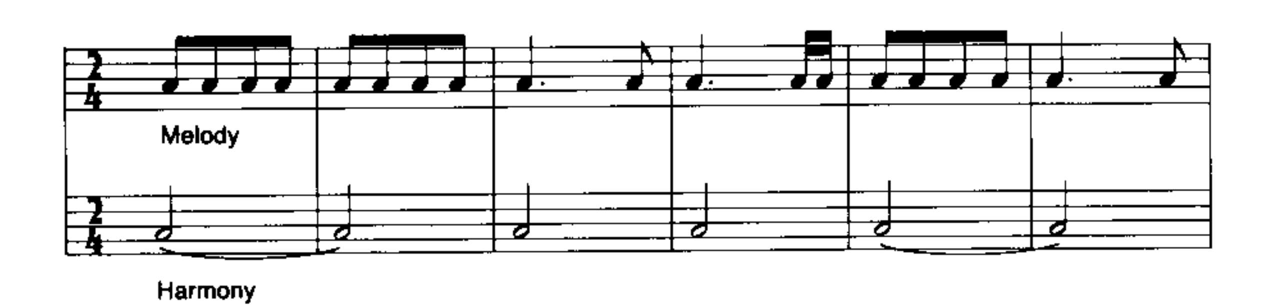
In principle it would seem that formal structure in the melody could be ensured by selecting and manipulating geometric shapes that repeat, reflect, expand, contract, and meet end-to-end through translation. Unfortunately, this is extremely difficult to achieve in practice because it requires that the curves be scaled and pieced together in such a way that their important features are not lost in the sampling process. It forces the user and the programmer to deal with many tedious ideas that relate only tangentially to music. Without intuitive control over the relation between rhythm and geometry, this technique holds limited promise for application to the MT idiom. Of course, there is a great potential here for anyone interested in exploring new directions. But I would hesitate to show all my results to anyone who doubts that computers can be used to compose "real" music.

A Numerical Approach

In order to get to the point where the melodization process begins, let's consider an actual situation where the following technique was used. A lyric was given and the rhythm for the entire melody was worked out by my usual method of transcribing its recitation into musical (rhythmic) notation and adjusting for meter and bar lines. Then, a chordal rhythm was superimposed on the melodic rhythm, i.e., I decided where successive harmonic changes would occur. The first eight measures then had the rhythmic definition shown in Fig. 12-3.

Next, a sequence of chords was selected to fill the harmonic rhythm. (We'll look into the problem of composing harmonic continuity in the next chapter.) The five structures chosen for the phrase shown below were all from the major-seventh class (marked **MA7** in the tables) with root tones E♭, A♭, F♯, E, and B in that order.

Now, a set of melodization vectors was chosen. By using a small number of vectors, each with no more than four entries, a certain amount of structural integrity is assured because these vectors (and the patterns they contain) must undergo some repetition to account for all the attacks. In APL the
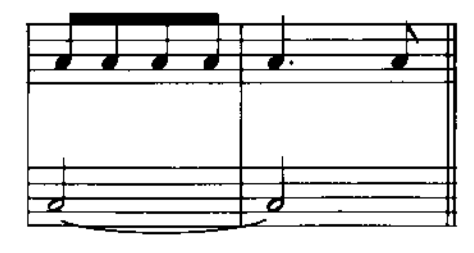




Fig. 12-3. Simultaneous rhythms of melody and harmony selected for a given lyric.

vectors that were selected could be expressed as:

M1 ← SCL[4 0 1 2]
M2 ← SCL[3 2 1 2]
M3 ← SCL[1 0]
M4 ← SCL[6 0]
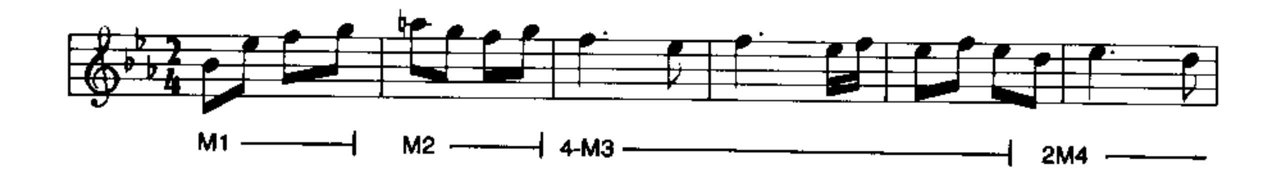
The indices here refer to the scale-like structures in Table 12-1. It is worth noting that the entries were chosen so each vector ends one scale step above or below the first entry in some vector!

Any method can be used to select the sequence in which these vectors should appear, but too often no consideration is given to the problem of connecting the successive segments of the melody. In this particular case, the method that was used depended on geometrical considerations that confined the melody to the area between two selected curves in pitch space. It is critical here to notice that these vectors contain indices to scales. Each change in harmony here changes the root tone of the scale, though in this case the scale intervals remain the same (because all the scales use the same MA7 structure).

An algorithm to control the range of the notes specified by the melodization vectors will seem quite complex if those vectors are interpreted in a rigid way with respect to the root tones. My approach applies the indices to a single root tone—the first. In this example, this method guided the selection process so that the entire melody was chosen as if only an E♭-major structure was to be melodized (Fig. 12-4). If the same choice of vectors had been applied directly to the proper sequence of chords (scales), the melody would have been the one shown in Fig. 12-5. The ungainliness apparent here as the melody "adjusts" to each root tone is a typical feature of direct melodization of harmony; the overall shape of the melody loses coherence because the individual segments are displaced from each other through seemingly arbitrary intervals.

Still, we must account for these changes of harmony in the final step of the process; the melody cannot remain in a single key. A musician would alter the monochordal melody to suit the correct harmony by leaving each note just where it is on the staff but adding sharps or flats as needed to account for the designated tonality at each point. This effectively maintains the geometric shape of each melodization vector even though the vector's entries are interpreted with respect to a different structure. It also preserves the "connected" quality of the original melody, i.e., the way each vector leads to the next through one "scale" step.

In our example, the first two measures need no alteration because the first chord is the one that was used throughout the initial melodization. In the third measure, the root tone of the harmony changes to A♭. The algorithm I used to mimic the musician's approach found that this was three scale
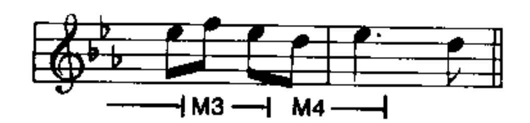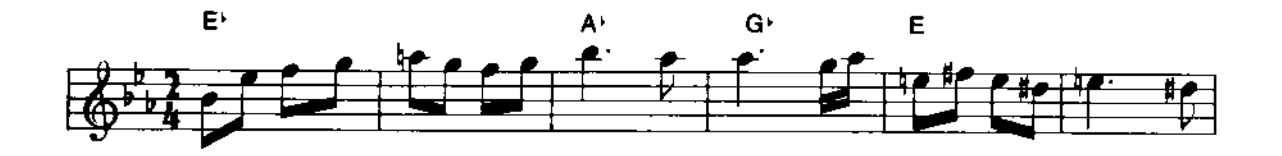




Fig. 12-4. The sequence of melodic vectors chosen for the melodic rhythm shown in Fig. 12-3, but fitting only a single harmonic structure throughout.
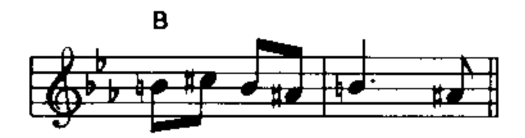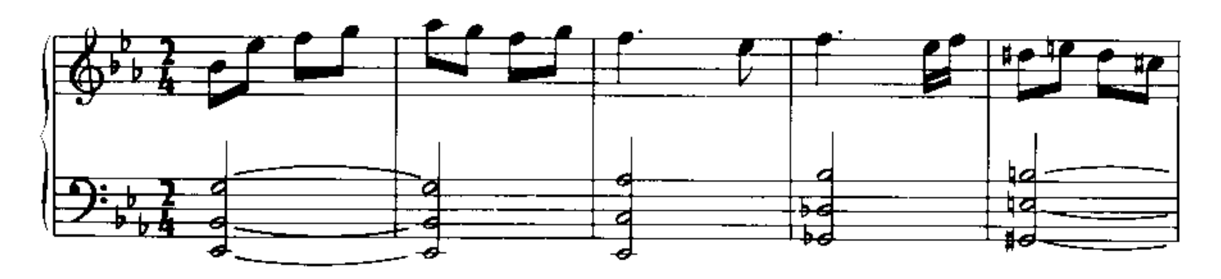




Fig. 12-5. The same sequence of melodic vectors adjusted to fit a sequence of scales satisfying the harmonic rhythm of Fig. 12-3.

steps above the initial root (in the initial scale). So *minus* three (modulo seven) was added to each entry in the melodization vector to compensate for this root-tone motion. Notice, the melodic fragment here contains the entries, 1 0. Adding minus three changes them to ⁻2 ⁻3 and, through modulo seven arithmetic, to 5 4 in the A♭ scale. The next root, F♯ or G♭, is not in the initial scale, but my algorithm decided it was *approximately* two scale steps above the initial root. As a result, minus two was the value used to adjust the melodic line for this tonality. Following this procedure throughout the piece produced, I think, a noticeable improvement (Fig. 12-6).

I have not shown any of the algorithms here because they were never fully implemented in APL.
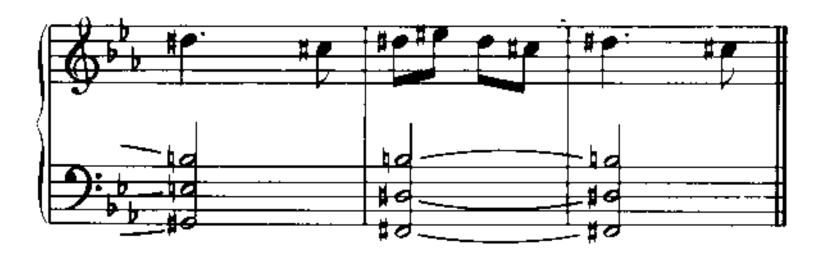




Fig. 12-6. The same sequences of melodic vectors and harmonic scales, but with root-tone-motion compensation to fit the melody to the harmony.

When I began to express them in the better language, the programming details that were no longer needed fell away, allowing me to see certain musical ideas that had been hidden. These ideas revealed a general approach to harmonization of melody that captured my imagination. From there, the path has never turned back to the simpler technique.

HARMONIZATION OF MELODY

This technique also presents us with a pair of problems. Not only must each chord fit the given melodic segment but, on going from one chord to the next, each change in tonality must seem appropriate. In the remainder of this chapter I'll demonstrate a technique that relies on the explicit evaluation of tension between melody and harmony to account for the problem of fitting the two together. The problem of continuity will be treated through the use of preference arrays.

When a composer selects a chord to harmonize a string of notes, he must (at least subconsciously) consider the types of chord from which he will choose and the root tones that will bring each chord into a position that forms an acceptable tonal relation with the melody. It would be unthinkable for him to make a list of each type of chord transposed into all twelve keys and then refer to that list to make his choice. Yet such a list would include all the cases he does consider as well as all the ones he misses through oversight or fatigue. A computer algorithm can not only assemble the list, it can search through it to find all the acceptable entries.

To construct the list, we can begin by building an array containing the intervals for each type of structure we may want to use. For example, a structure array defining just major and minor triads would be

```
SA ← 2 2 ⍴ 4 3 3 4
```

The first row, **SA[0;]**, would hold the intervals of the major triad, 4 3, and the second, **SA[1;]** would contain the minor intervals, 3 4. Now, the function listed in Fig. 12-7 will create an array containing all of the types of chords defined in the given structure array (SA) built on all the (root) tones in our pitch system.

This function, **HBLK**, will produce an array (a Harmonic-structure BLocK) in which each row describes one of the chords from the interval-structure array (SA) built on a particular root tone. The first entry in a row specifies the type of chord by showing its row index in SA, i.e., the entry *r* would refer to the structure in **SA[*r*;]**. The remaining entries in that row display the pitches in the chord. Notice that the first two entries in a row then give the structure and root and so are sufficient to identify the chord to the composer. Of course, two numbers, such as 1 10, may identify a B♭-minor chord sufficiently, but they don't do so efficiently. However, it's easy enough to construct an array of names that "parallels" the structure array, SA:

```
NAMES ← 2 3 ⍴ 'MAJMIN'
```

and a function that makes use of the numbers (1 and 10) and the fact that

```
(N2NP 10), NAMES[1;]
```

will spell out "**A ↑ MIN.**"

```
     ∇ Z←HBLK SA
[1]    Z←(TONSYS,⍴SA)⍴SA
[2]    Z←+\(2 1 0 ⍉(1,(1↑⍴SA),TONSYS)⍴⍳TONSYS),Z
[3]    Z←(2 0 1 ⍉(1,TONSYS,1↑⍴SA)⍴⍳1↑⍴SA),TONSYS|Z
[4]    Z←((×/2↑⍴Z),¯1↑⍴Z)⍴Z
     ∇
```

Fig. 12-7. Function HBLK constructs all possible chords that conform to the given structure array, SA.

The "harmony block" (the array produced by **HBLK**) can also be tested row-by-row against the target melody to find the most suitable harmonization according to some specific criterion. That is, if we call HA the array:

HA ← HBLK SA

and say TM is the vector of pitches in the target melody, then we might decide to count the number of melody notes present in each of the chords (rows) of HA:

CNT ← +/(0 1↓HA)∊UNIQ TONSYS|TM.

CNT would be a vector whose Nth entry showed the number of melody notes contained in the chord in **1↓HA[N;]**. (Note: the "one drop" here removes the chord type so that only the notes are used in the test.) This vector could now be used in a conditional expression to eliminate rows of HA whose "count" falls outside any desirable range. For example,

HA ← (CNT ≥ N/[0] HA

would remove rows with fewer than N melodic tones in the chord. Any number of other criteria can be used to test the chords against the target melody and compress out those chords that fall below some explicit standard.

But rather than discuss one abstract point after another without reference to any particular musical situation. I'll demonstrate the use of one of my earlier harmonization algorithms in treating the one melody in Figure 9-9 that was not exhibited with harmony in the chapter on melody. The main func-

```
    ∇ A HOFM M;I;S;P,F;HB;HA;CNT,TM,XF;XP;CND;TGM
      HB←HBLK SA
      H←((ρA),2+¯1↑ρSA)ρ¯1
      I←0
     L0:S←⍳1↑ρSA
      XF←F←⍳1+¯1↑ρSA
      'I=',(⍕I),' ?'
      →(L0,L0,L0)[HOFMIN 0]
     L1:TM←TONSYS|M[(-⌈A[I])↑⍳(+\⌈A)[I]]
      P←TM
      XP←¯1
      'TM=',⍕TM
      'S, P, XP, F OR XF (OR E) ?'
      →(L0,L1,L1)[HOFMIN 1]
     L2:HA←(HB[;0]∊S)/[0] HB
      HA←((∨/(0 1 ↓HA)[;,F]∊P)∧∧/~(0 1 ↓HA)[;,XF]∊XP)/[0] HA
      CND←⌈/CNT←+/(0 1 ↓HA)∊UNIQ TONSYS|P,TM
      →(0<1↑ρHA)↑L3
      'NO STRUCTURES POSSIBLE'
      →L0
     L3:'CND=⌈/CNT?  (CND,CNT,HA,H?)'
      →(L0,L1,L2)[HOFMIN 2]
      H[I;]←HA[CNT⍳CND;]
      →((ρA)>I←H[;0]⍳¯1)↑L0
      'OK?'
      →(L0,L0,L0)[HOFMIN 0]
      HOFMOUT⍳ρA
    ∇
```

Fig. 12-8. Function HOFM interactively guides the user in the harmonization of a given melody.

tion is called **HOFM** for Harmonization OF Melody (Fig. 12-8). In addition to **HBLK**, this function also requires three auxiliary routines: **HOFMIN**, **HOFMOUT**, and **SHOW** (Fig. 12-9).

Preliminaries to the Harmonization Algorithm

As was true for the function **FORM**, this one also serves mainly to keep account of bothersome details analogous to bookkeeping. The user who understands certain cryptic messages and knows the internal names of critical global variables is relatively free to concentrate on his own criteria for harmonization. In this first example, tension will be given the starring role in fitting harmony to a given melody. The melody to be harmonized was created in Fig. 9-9 using

```
4    DVLP2    2 7
```

In keeping with the austerity practiced in designing the function **DVLP2**, we'll limit ourselves to the structure array SA shown above which contains only major and minor triads. Through the use of tension, we can find particular chords from this set that will harmonize the segments of the melody, but we must also exert some control over the relationship between successive chords in the continuity. Without yet looking more deeply into that problem, we'll make use of *preference arrays*. As an example, one of these arrays will be:

```
PRF1←4 3⍴0 1 5 0 1 6 1 0 ¯1 1 0 5
```

or

0	1	5
0	1	6
1	0	-1
1	0	5

Because SA describes just two structures, every chord will correspond to type (row of SA) zero or one. The entries in the first two columns of our preference arrays will always refer to types of structures in this way. These first two columns relate the previous chordal type to the one about to be selected. The third column in the preference arrays specifies the motion of root tones. **PRF1** can now be restated in the form of a compact set of guidelines:

Row 1: If the current chord is major (row zero of SA), the next chord can be minor (row one of SA) providing the root tone for the new chord is five semitones above the present root.

Row 2: A major chord (zero) can also go to a minor chord (one) if the root tone moves up six semitones.

Row 3: If the present chord is minor (one), the next chord can be major (zero) if its root is one semitone below (minus one) the current root.

Row 4: A minor chord (one) may also move to a major one (zero) if the root moves up five semitones.

There isn't much slack in these guidelines; situations might arise in which no chord can be found that will satisfy one of these rules. The analysis routines should inform the user when such a situation arises so that he can suggest a less stringent condition that might be met. At the other extreme, more than one result may satisfy the given conditions. In that case, the user may decide to let the program choose among the possibilities in some prearranged way, or he may want to enter additional constraints of his own. Even if no such problems arise, the tautness of these guidelines may harness the results in an overbearingly strict style. To get around this obstacle, a global name, PREF, is set aside for the preference array within the algorithm. The user can begin by setting

```
PREF←PRF1
```

to invoke the preference array defined above. If at any time he feels his results are falling into a stale pattern, he can revive them with fresh guidelines by resetting PREF to some other array, such as

```
PREF←PRF2
```

```
      ∇ Z←HOFMIN J;X
[1]     Z←⍳0
[2]    L1:X←⍞
[3]     →(0=ρ,X)↑0
[4]     →('?'=1↑X)/(L10,L11,L12)[J]
[5]     →('⎕'=1↑X)↑L9
[6]     ⍎X
[7]     →L1
[8]    L9:H[⍳I;] SHOW 1↓X
[9]     →L1
[10] L10:'ENTER CR OR ''I←SOME VALID ATTACK INDEX'''
[11]    →L1
[12] L11:'CR OR SET EXPLICITLY ANY OF THESE:'
[13]   '       S IMPLIES STRUCTURE TYPES TO BE CONSIDERED'
[14]   '       P EXPLICIT CHARS TO BE TREATED    (DEFAULT P = TM!)'
[15]   '       F EXPLICIT FUNCTIONS FOR THE P'
[16]   '       XP EXCEPTIONAL CHARS TO BE AVOIDED'
[17]   '       XF FUNCTIONS FOR XP TO AVOID (DEFAULT = ALL)'
[18]    →L1
[19] L12 'CR OR RESET CND TO SOME CNT ENTRY FOR USE AS CNT⍳CND'
[20]   '       NOTE: CNT, HA, AND H MAY BE MANIPULATED AS NEEDED'
[21]   '       HA CONTAINS THE STRUCTURES FOUND OK FOR THIS I'
[22]   '       H IS THE FINAL CHORDAL ARRAY'
[23]   '        (WITH ¯1 ELEMENTS WHERE NOT YET DEFINED)'
[24]    →L1
      ∇


      ∇ HOFMOUT I;J;MM
[1]     MM←M
[2]     J←0
[3]    L1:(1 3 ρ'M:  '),N2P A[(,I)[J]]↑MM
[4]     (1 7 ρ'H:      '),N2NP(0 1 ↓H)[(,I)[J];]
[5]     MM←A[(,I)[J]]↓MM
[6]     ''
[7]     ''
[8]     →((ρ,I)>J←J+1)↑L1
[9]     'TO KEEP H, ENTER: ''NAME←H'', OTHERWISE CR'
[10]    ⍎⍞
      ∇


      ∇ A SHOW X
[1]     →(0≠ρ,X)/L1
[2]     A
[3]     →0
[4]    L1:⍎X
      ∇
```

Fig. 12-9. Function HOFMIN handles the interactive input to HOFM, function HOFMOUT constructs the final listing of results, and function SHOW displays "quad-requested" items for HOFMIN.

We would now be ready to call the harmonization function, HOFM, if we had the left and right arguments at hand. The right argument is the melody to be harmonized, and we will use the variable M after setting it by the statement

M ← 4 DVLP2 2 7.

The left argument deserves a lengthier introduction.

The Attack Vector for Harmonization

The left argument must describe how the melody is to be segmented for harmonization. In other words, it must specify how many melodic pitches are to be accompanied by the first chord, how many must fit the second, and so on. This means it will be a numeric vector whose entries must satisfy the condition:

(+/A) = ⍴M

that is, the sum over the entries in the left argument (the attack vector) should equal the number of entries in the right argument (the melody). Frequently I will refer to the left argument as the *harmonic* attack vector although, strictly speaking, it is a *melodic* attack vector in which the individual entries apply to separate chords instead of to time intervals.

Each entry in the left argument must be a positive number not necessarily an integer, although the sum of the entries must still be an integer in order to account exactly for all entries in the melody. Two consecutive attacks of 2 3 would mean that two notes of the melody must go with the first chord and the next three notes in the melody must fit the second chord. If we changed the two attacks to 2.5 □ 2.5, the same five notes would be harmonized by two chords, but the change in harmony would occur while the third melodic tone was still being played. The actual value of the decimal doesn't matter; the first 2.5 indicates the first three notes are to be harmonized, but the chord will change somewhere during the time allotted to the third note. The second 2.5 also says three notes are to be harmonized, but the previous note, because it is still being played, counts as one of them. Similarly a sequence of attacks such as .3 .3 .3 .1 would cause four successive chords to harmonize the same pitch. Again, the actual values have no effect other than to change the pitch when the running total reaches a new integer.

In this example we'll avoid unnecessary complexity by using only integers in the attack vector. But as you might expect, because we are again dealing with rhythmic matters, no attempt will be made to automate the construction of this vector. I'll try to describe here the considerations that led to the attack vector used in this example.

First, the melody had been derived through "building blocks" that made no direct reference to scales or even scale kernals. It's not surprising than that the pitches in M contain two potentially troublesome chromatic successions of tones, 4 5 6 7 and 7 8 9 10 11. Such lengthy sequences of semitones present a severe challenge to any harmonization logic. Few, if any, chords will fit such segments within given tension limits and at the same time fit into the continuity, following the previous chord according to the requirements of preference arrays or "sensibility." A glance at the keyboard diagrams in the preceding chapter that show the tension of all tones with respect to major and minor triads will point out that, without changing chords, sequences of three or more consecutive semitones are almost sure to contain some "blue" notes. (In the denominations of tension currency, the blue ones are worth a thousand.) So the surest approach is to assign values to the attack vector that force the harmony to change in the midst of such a sequence so as to allow no more than two or three consecutive semitones in the same melodic segment.

The second consideration relates to temporal form. In this case, the 55 notes of the melody had been distributed in metric phrases beforehand. Recall, there were four examples that had this same number of melodic tones. In each piece, the same rhythm was used in which the 55 notes were stretched over five four-bar phrases as follows:

2	4	4	2
2	4	4	2
2	4	4	2
4	4	4	4
1	1	1	0

Here the melodic phrases suggested by the building blocks were interesting enough to be used directly for the rhythmic phrases, without interference between the two. (You may recall that the building blocks in the original example in Chapter nine created melodic segments that seemed so trite, interference was used there to "rephrase" the melody.) Before harmony was even considered in this case, the phrasing caused those sequences of semitones to straddle the phrase boundaries where chord changes were sure to occur anyway, thus practically eliminating the problem of consecutive semitones.

The left argument was constructed in the statement

```
ATK←2 4 4 2 2 4 4 1 1 2 4 4 2 8 8 1 1 1
```

This attack vector has 18 entries. It indicates the first chord chosen must harmonize the first two notes of the melody. The second chord must fit the next four entries in M, and so on. The eighteenth chord will be chosen to harmonize the last of the 55 notes in M.

Using rather inexpensive hardware—a pencil—directly on a listing of the pitch numbers in the melody, one can mark off the successive tones as they are included in the harmonic attack vector. In this way, the first three phrases were accounted for, almost copying the melodic attack numnbers shown above directly into the harmonic attack vector. My normal approach is to limit each harmonic attack so that no more than three to five different pitches will need to be harmonized by one chord. The melodic attack groups for the 55 tones here meet that concern directly: no group has more than four notes.

But on reaching the fourth phrase, the one containing four fours, my pencil showed that five notes repeat in sequence, accounting for all 16 attacks. Instead of the threat of having too many notes to be harmonized by one chord, this raises the possibility that there will be too few chords that will harmonize the same repeated tones. If we assign each group of four notes to a chord, almost the same set of possible harmonizations will be found for all four chords. With only two kinds of structures (major and minor triads) from which to choose, all selections would most likely appear to be from the same overall tonality, and the resulting phrase would require very special treatment to avoid the uncomfortable harmonic sensation of "going nowhere" or even "backtracking." For this reason, two chords accounting for eight notes each were indicated.

Parameters and Phases of Execution

The listing of the entire session to be described is shown in Listing 12-1 at the end of the chapter. For reference purposes, the significant data just described are exhibited before the actual call for execution of the function **HOFM**. Even my pencil markings, breaking up the M (melody) vector for harmonization and highlighting the chromatic pitch sequences, can be seen. Notice also the definition of the alternate preference array, PRF2.

When at last we hoist anchor, calling the function with explicit arguments,

```
ATK    HOFM    M
```

the computer fires back the enigmatic question

```
I = 0 ?
```

To put the question in more user-friendly terms, the computer is merely asking if it would be alright to begin at the beginning with the attack group whose index (I) is zero. If I were so motivated, I could type, "I←6," at this point to force the seventh chord (whose index is 6) to be selected first. Lacking such enthusiasm, I simply press the Return key to indicate that "I=0" is fine with me.

Now the system responds with another user-antagonistic message:

```
TM = 4 6
S, P, XP, F OR XF (OR ⎕ ) ?
```

The first line states that the target melody (TM) for the chord about to be selected contains the two pitches, 4 and 6. The next line asks whether I care to assign special values to any of the parameters listed (S, P, XP, F, or XF) or see the results of harmonization so far (which could be requested by typing a quad symbol). "Why should it offer to show results that don't even exist because harmonization hasn't begun yet?" you ask. Claiming economy rather than sloth, I programmed it to display the same message for every segment of harmonization, including the first. By responding

```
S←F←1
```

I ensure that this harmonic structure (S) will be of the type in row one (1) of the structure array and that at least one of the pitches in the target melody will be note number one (the pitch above the root) in the chord. If S is not specified, all rows of the structure array will be considered in choosing a chord.

The symbol F conforms to Schillinger's terminology wherein chordal tones are referred to as harmonic *functions*. [I would never have used that symbol had I known that someday I might need to make clear to others the useful function served by a harmonic function in a mathematical function expressed by an APL function. As part of his insight into tension, Schillinger felt that, in a harmonic structure built in thirds, each higher note introduced into the chord serves the function of adding increasing tension. Thus he believed the numeric value indicating the position of a tone in a chord (e.g., the fifth, the seventh, the ninth, etc.) "functioned" in some way as a measure of tension. In the system of measurement adopted here, this does not hold.] In this algorithm, F merely serves as the index of a note in a harmonic structure, and with only triads defined in the structure array, F may only have the values zero, one, and/or two. By default, if F is not specified by the user, a chord will be tentatively acceptable if it contains any of the target tones in any indexed position.

The default value for the parameter P is a vector equal to the target melody. The program tests all chords produced by HBLK and keeps only those that have any of the pitches in P in any of the chordal positions indexed by F. The user may alter P in any way he wishes. The parameters XP and XF can be used to identify pitches that must *not* appear in particular chordal positions. These exceptional pitches (XP) are initialized to ¯1, a value which will never appear in any chord because the tones are specified modulo 12. XF is preset along with F to contain all chordal positions. So by default, no exceptions will be found. The user might, for example, set XP to 3 and XF to 0 2 to prevent any chord from having E♭ either as its root (index 0) or fifth (index 2).

After another carrriage return (an empty line) to signal that I don't care to specify other parameters, the program begins the last phase of its selection process for this chord by presenting one more cryptogram:

```
CND=⌈/CNT?   (CND,CNT,HA,H?)
```

Deciphered, this asks, "Will it be acceptable to choose the 'best' chord to be the one satisfying the condition (CND) that it possesses by direct count (CNT) the largest (⌈/) selection of pitches from the target melody and the parameter P? Also, would you care to examine the current value of the condition (CND), the count (the vector CNT), the harmonic alternatives (HA) which are the chord structures from which the final choice will be made, or the list of harmonic structures (H) already chosen thus far?"

By responding with another empty line, I indicate my answers to be "yes" and "no" respectively. (In the interest of clarity, I should point out that these are not really questions in the "yes-or-no" sense. They are actually reminders which the user can choose to ignore.)

Throughout the harmonic selection process, the program prompts the user with the same three

questions for each chord:

```
I = N ?
S,P, XP, . . . ?
CND = ⌊/CNT? . . . ?
```

In each case, the user is given the chance to alter the various parameters or conditions which will otherwise be chosen according to the preprogrammed logic. The three questions initiate three distinct phases of the selection process. The user can enter as many statements as desired in each phase to set parameters and examine existing values. Only after he enters an empty line are the current values of all parameters pertinent to that phase passed through the logical operations in the program. Then the question that initiates the next phase of the operation is displayed. For brevity, the three phases will henceforth be referred to as "I-time," "S-time," and "C-time," respectively.

I-time merely establishes the index, I, for the chord that is to be chosen. By default, I is set for the chord with the lowest index that has not yet been defined. When S-time begins, the parameters (S, P, F, XP, XF) and the current target melody (TM) are reset by the program. The user then may alter any of them before going on to C-time. On entering C-time, just those harmonic structures that satisfy the parameters are made available and the "score" for each one is tabulated. The user may then impose further selection criteria to narrow down the list of alternative structures still further. The first chord that satisfies the current condition (CND) is then selected.

Using the Tension within the Harmonization Algorithm

For the next six chords (I-values 1 through 6, no specifications were made during I- or S-times. But at each C-time, the statement

```
0 TPRF 1000
```

was entered. This makes the block of eligible structures undergo further sifting according to tension considerations. The values given here (0 and 1000) define the allowable range in which the total tension of the melody must fall. That is to say, the tension of the target melody, taken as a whole with respect to an acceptable chord, must lie between the given limits. Figure 12-10 contains listings of the functions **TPRF** and **TCNT**. Note that **TPRF** calls **TCNT** and the latter function changes the meaning of the vector, CNT, from a count of "target-melody hits" in each chord to a list of tension values of the target melody with respect to each chord. Function **CYCLE**, which is called in statement **[4]** of **TPRF**, will be described shortly. A special output function, ZOUT, was used here to display messages and also keep statistics for further study of tension. For our purposes here, in statements **TPRF[8]** and **TCNT[7,12]** the characters "**0 ZOUT**" can be ignored in the listings, leaving just the "bare messages" in those statements.)

If the conditions described in the preference array, PREF, can't be satisfied or if all the structures that do conform to those conditions fail to meet the tension requirements, an appropriate "error" message will be displayed. The user can then alter any parameters or preference specifications at will until the message appears

```
n STRUCTURES.     NOW CND = ⌊/CNT?
```

Here *n* will be the number of eligible structures found. The message goes on to say that the condition for final selection will isolate the chord with the smallest melodic tension value in CNT (unless the user wishes to take some other action). In five of the six cases (the exception being $I = 3$) the message tells us that specificaitons in the PREF array and in the tension limits for the call to **TPRF** eliminated all but a single possible harmonization. For the third case, we're told that two harmonic candidates satisfied all the requirements for structure, root tone, and melodic tension.

Additional Controls in Harmonization

At S-time for the eighth chord in the sequence ($I = 7$), a quad symbol was entered, causing the pro-

```
      ∇ MN TPRF MX;K
[1]     K←(PREF[;0]=H[I-1;0])/ι1↑ρPREF
[2]     →(0=ρK)↑L0
[3]     S←PREF[K;1]
[4]     CYCLE PREF[K;2]
[5]     →(2>J)↑0
[6]     K←(HB[;0]∘.=S)∧HB[;1]∘.=P
[7]     →(0≠∨/∨/K)↑L1
[8]   L0:0 ZOUT 'NO STRUCTURES CONFORM TO PREF'
[9]     →0
[10]  L1:HA←(∨/K)/[0] HB
[11]    CNT←+/(0 1 ↓HA)∈UNIQ TONSYS|P,TM
[12]    HA←HA[K←⍋(,K)/,(ρK)ριρS;]
[13]    MN TCNT MX
      ∇
```

```
      ∇ MN TCNT MX;K;T
[1]     Z←''
[2]     T←(ρCNT)ρK←0
[3]     L1:T[K]←(UTNSN 1↓HA[K;],TM)-(UTNSN 1↓HA[K;])+UTNSN(~TM∈1↓
HA[K;])/TM
[4]     →((ρCNT)>K←K+1)↑L1
[5]     K←(T≥MN)∧T≤MX
[6]     →(0<+/K)↑L2
[7]     0 ZOUT 'NO H IN THIS TNSN RANGE'
[8]     →0
[9]   L2:HA←K/[0] HA
[10]    CNT←K/T
[11]    CND←⌊/CNT
[12]    0 ZOUT(⍕+/K),' STRUCTURES.    NOW CND=⌊/CNT ?'
      ∇
```

Fig. 12-10. Functions TPRF and TCNT isolate chords that conform to a given preference array and tension range.

gram to display the seven chords selected up to this point:

1	1	4	8
0	0	4	7
1	6	9	1
0	11	3	6
1	4	7	11
0	3	7	10
1	9	0	4

These are the products of the **HBLK** function that survived all the steps in the selection process for each of the first seven segments of the melody. Remember, the first element in each row points to a row in the structure array (SA), and so tells whether this chord is major or minor. The three chordal tones follow with the root as the second entry in each row.

Reading down the first column, we see that successive chords alternate repetitively between the two types as dictated by the preference array. Reading down the second column (but "between the lines" in a mathematical sense), we observe the

beginnings of a less obvious sequence. Successive root-tone intervals appear to be falling into the pattern: down one semitone, up six, up five, and up five more. This is a typical product of such a limited preference array.

Such sequences are permissible and often desirable in the MT idiom when they are synchronized with the melodic phrasing. Had I let the harmonic attack vector duplicate the melodic one, I might have let the sequence continue here. But notice that in the second phrase, instead of assigning four chords to harmonize successively two, four, four, and two melodic tones, that last pair of notes is to be broken so that each tone is harmonized independently. This destroys the synchronization between the harmony and the melodic phrasing. So, to make a complete break with the sequence, I called for a root-tone motion that doesn't appear in the preference array:

```
CYCLE ¯3.
```

This forces the root tone to move down three semitones by setting the F and P parameters accordingly.

Upon entering C-time, a display of the harmonic alternatives was requested (by typing HA). As expected, since the last chord in the sequence was built on pitch number nine, here we find both kinds of triads with a root tone of six satisfy the requirements. Entering CNT shows that both structures score a single hit in meeting the target specifictions. The "target" here contains the melodic pitch eight and a root tone three semitones below the previous root. Because compliance with the root-tone specification was forced, a count of one implies that the melody here is not in either chord. So the **TCNT** routine is called next to eliminate chords creating tension outside the range from zero to a thousand and also to reset the CNT variable to the corresponding tension values.

Notice (that the call here is directly to **TCNT**, not to **TPRF**. The latter function would try to satisfy the instructions coded in the preference array before calling **TCNT** itself. In specifying **CYCLE ¯3** at S-time, I caused all structures that could have satisfied the requirements in PREF to be removed, so all we want at this point is to control the tension range. The message produced by **TCNT** shows that neither function has been eliminated, but their tensions can now be seen (after typing CNT) to be eleven for the major triad and 110 for the minor. By setting the condition (CND) to the greater (rather than the lesser and default) value:

```
CND ← ⌈/CNT
```

the minor triad will be selected. Not only does this disturb the pattern of root tones, it also jogs the sequence of structures, 1 0 1 0 . . . , from its monotonous (or should I say "bitonous") course.

For the next chord (I = 8), there is again a single target pitch (TM = 9). At S-time, the statement

```
0 FCHNG 2
```

causes the root (function zero) of the previous chord to remain unchanged as a pitch, but to undergo a Functional ChaNGe to become chordal tone number two in the new harmony. We see at C-time (again by typing HA) that the choice is between a B-major and a B-minor structure. By entering only an empty line, we let the program decide which one will be used.

At I-time for the next chord (I = 9), the preference array itself is reset to PRF2 to introduce a more varied set of rules relating potential root-tone motions and structure sequencing. And again at C-time, the function **TPRF** is invoked to combine the new selection rules with tension limits of zero to a thousand.

Controlling Climax with Tension

Introducing a high point in the tension creates an esthetic climax. With 18 chords to be specified in the harmonization, the eleventh chord (I = 10) is well positioned (just over the halfway point) to launch a climax. To show how easily a climax can be introduced in a traditional place (not to mention how easily traditional rules and tastes can be violated), the tension limits at C-time for this chord are raised:

```
1000 TPRF 2000.
```

Again, only a single structure is found that satisfies all the conditions. The tension it creates with the melody is shown to be 1110, indicating the two unique tones in the target melody (three and ten) manage to generate a minor second, a major seventh, and a major second with respect to the chord.

My only direction for the twelfth chord (I = 11) comes at C-time with

```
100 TPRF 1000.
```

This is to let the tension recede by no more than one (logarithmic) step from its climactic level. And once more, the computer announces that one qualifying structure exists.

With the following chord (I = 12), I resume the C-time specification

```
0 TPRF 1000,
```

but now I am informed that three different structures fill all requirements. I ask to see the tensions associated with them by typing CNT. If there were significant differences between the three values, more information would be analysed until I could be convinced that one particular value has more appeal than the others. But the values (100, 110, 110) show a difference of only ten, and two of the values are the same, so further discrimination seems pointless—let the minimum tension be selected automatically.

Responding to the same treatment, chord number 13 comes through with no rivals.

```
0 TPRF 1000
1 STRUCTURES.     NOW CND=⌊/CNT ?
```

A mixture of curiosity and concern makes me ask to see the tension here:

```
CNT
121
```

My concern is over the need for a reasonable cadence in the few remaining chords; my curiosity is always aroused when a single chord fits all five unique pitches in an eight-note target melody. The tension value, 121, strikes me as comfortably low, but I recall that I chose the attack vector so that eight more target tones are coming up for the next harmonization and I don't want an anticlimactic letdown of tension just yet. So, for I = 14 at C-time, I insist that the tension not be allowed to drop below 100, and fortunately one structure is found that fills the complete bill.

Cadence Control

Having kept an eye on the complete melody vector, M, and its penciled groupings for harmonization, I see the potentially troublesome time has come for a cadence of three cords, each harmonizing a single note of the melody. At a point such as this, a composer takes careful aim at a tonality that will convey to a listener the sense of coming to a close. If he hasn't already decided on this tonality, he does so in typically human (i.e., desultory) ways, considering where the music is "coming from" and where it could "sensibly go." My early attempts to capture the logic of such situations caused me to construct ancillary functions, such as **CYCLE** and **FGHNG**, which were used above but not fully described. They and another function, **FDOWN**, are shown in Fig. 12-11.

These functions grew out of the idea that conventional sequences of chords often impress themselves on a composer's mind by the way particular tones of one chord move to tones in the next chord. Though redundant in function, these routines are unique in approach. That is, **CYCLE 5** and **0 FCHNG 2** will select exactly the same structures, but when the user calls the first one he is thinking in terms of a root-tone motion. When he invokes the second, his mind is focused on the way a particular voice leading occurs. Unfortunately, there is a dangerous use of "hidden parameters" in these programs (which I'll explain later) that I had forgotten about until long after this particular terminal session, although I was aware of an intermittent problem that I could not pin down.

```
      ∇ FOLD FCHNG FNEW
[1]     F←FNEW
[2]     P←H[I-1;1+FOLD]
      ∇



      ∇ FOLD FDOWN FNEW
[1]     F←FNEW
[2]     P←TONSYS|H[I-1;1+FOLD]-1,2
      ∇



      ∇ CYCLE N
[1]     F←0
[2]     P←TONSYS|H[I-1;1]+N
      ∇
```

Fig. 12-11. Functions FCHNG, FDOWN, and CYCLE can be called interactively to restrict the harmonic alternatives to chords that differ in particular ways from the previous chord.

Forgetting the possibility that the "problem" might occur, for the first chord of the cadence (I = 15) at S-time, I enter

0 FCHNG 1

specifying that the root (function zero) of the last chord is to be the third (function one) of this one. For the second chord (I = 16).

1 FDOWN 0

at S-time, calls for that same tone (the third of the last chord or function one) to move down one scale-like step (either one or two semitones) to become the root (function zero) for this chord. Then at C-time, on asking to see how many different chords have been found (**ρHA**), I find there are four—the reply "4 4" implies four structures with four entries in each, a structure index and three pitches. Presumably, these are the major and minor triads whose root tones are one and two semitones below the third of the previous chord.

This makes me aware that the "problem" has occurred, but I'm too concerned with *this* chord to remember that it could also have happened in selecting the previous one. (Luckily, it didn't.) My goal is to force a close with the lowest possible tension, so a range of zero to 100 is given. I'm told that two structures will work here, I find their tensions with respect to the target melody are eleven and zero, and I ask to see the harmonic alternatives. No further interaction is required if I really want the lowest tension, so a carriage return brings me to the last chord.

Setting S equal to zero commands the use of a major triad, and forcing F to be two makes the target pitch the fifth of the chord. At the close of C-time for this chord, the message "**OK?**" appears to inform me that the whole melody has been harmonized and, if I have any second thoughts, this is my last chance to return to any particular segment (by resetting the value of I). A final carriage return causes the program to list the entire set of results, pairing consecutive melody (M) and harmony (H) segments.

The Hidden Parameter Problem

The "problem" mentioned above occurs when the user forgets just how the auxiliary functions work and the way **HOFM** treats the parameters P and F in combination with the target melody, TM. In designing **HOFM** I wanted the user to have as much control over a special pitch vector, P, as possible. Remember, P is set to contain TM by default (see statement **[9]**). The composer is free to change this setting at S-time to make chord selection dependent on any desired relation between pitch and function. The default setting for the F (function) vector can be seen in statement **[5]**: all chordal tone indices are used. In addition, the "exceptional" pitch and function vectors (XP and XF) allow specific pitch-to-function relations to be avoided in the selection process.

Now an auxiliary function such as **FCHNG** resets P and F according to the arguments in its header. As a result, P no longer will contain any pitches from the target melody unless they happen

to satisfy the conditions programmed into the auxiliary function. At the close of S-time (signalled by the user's empty line), statement [14] in **HOFM** creates a set of harmonic alternatives (HA) containing all the chords produced by **HBLK** whose structure index is in the S (structure) vector. The next statement ([15]) narrows down this set of structures to those that satisfy the conditions relating F to P and XF to XP. Specifically, for any chord to be acceptable, at least one entry in P must appear in one of the allowed chordal positions, F, and no entry in XP may be in any of the forbidden positions, XF. At this point there is no quarantee that any pitch in the target melody, TM, will be in any of the remaining chords! Statement [16] then tallies the score achieved by each structure in hitting a
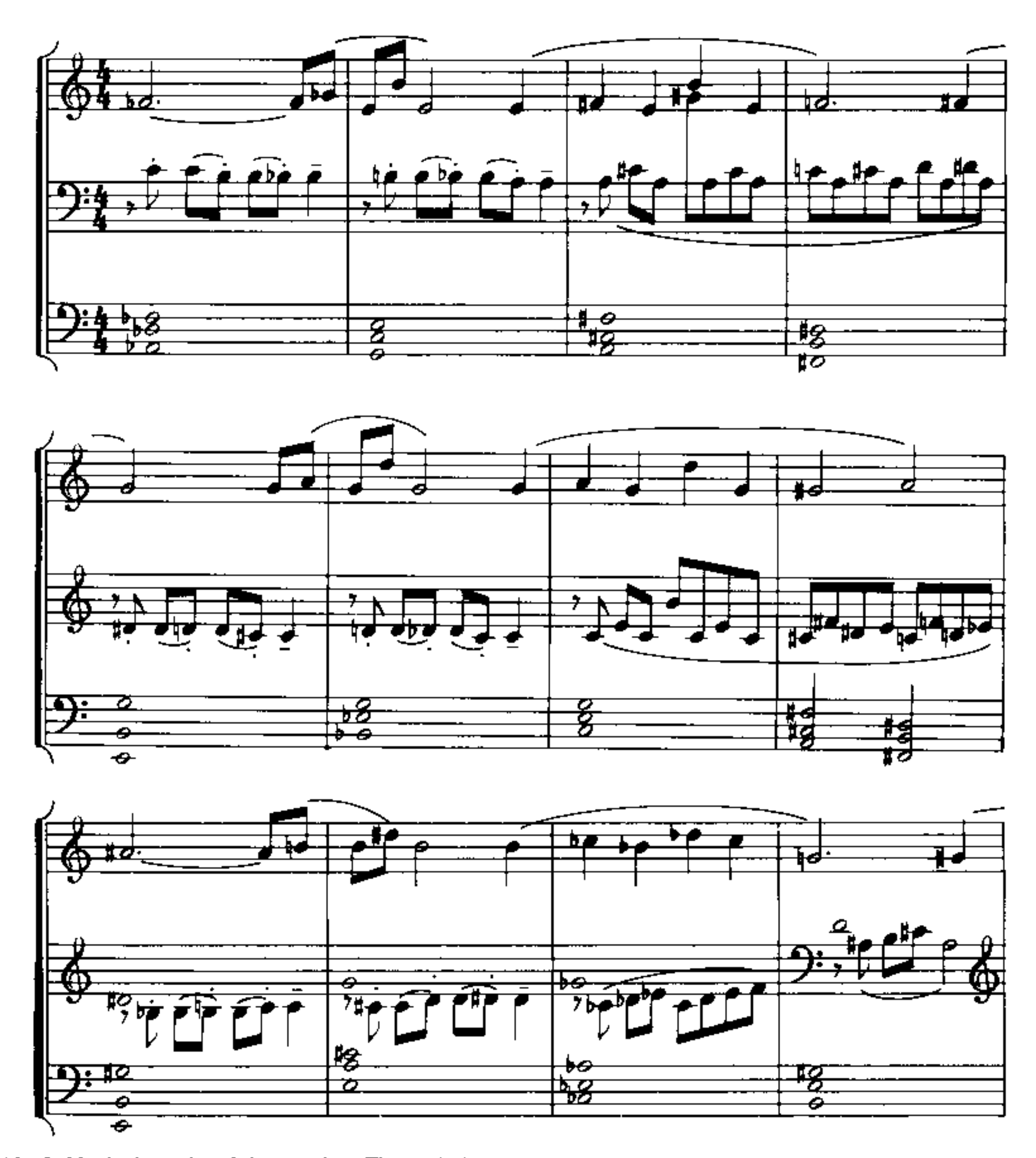


Fig. 12-12. Musical results of the session. The melody has been given durations, the harmony has been voiced, and a middle staff has been introduced to suggest additional, background material (© 1978 Jaxitron).
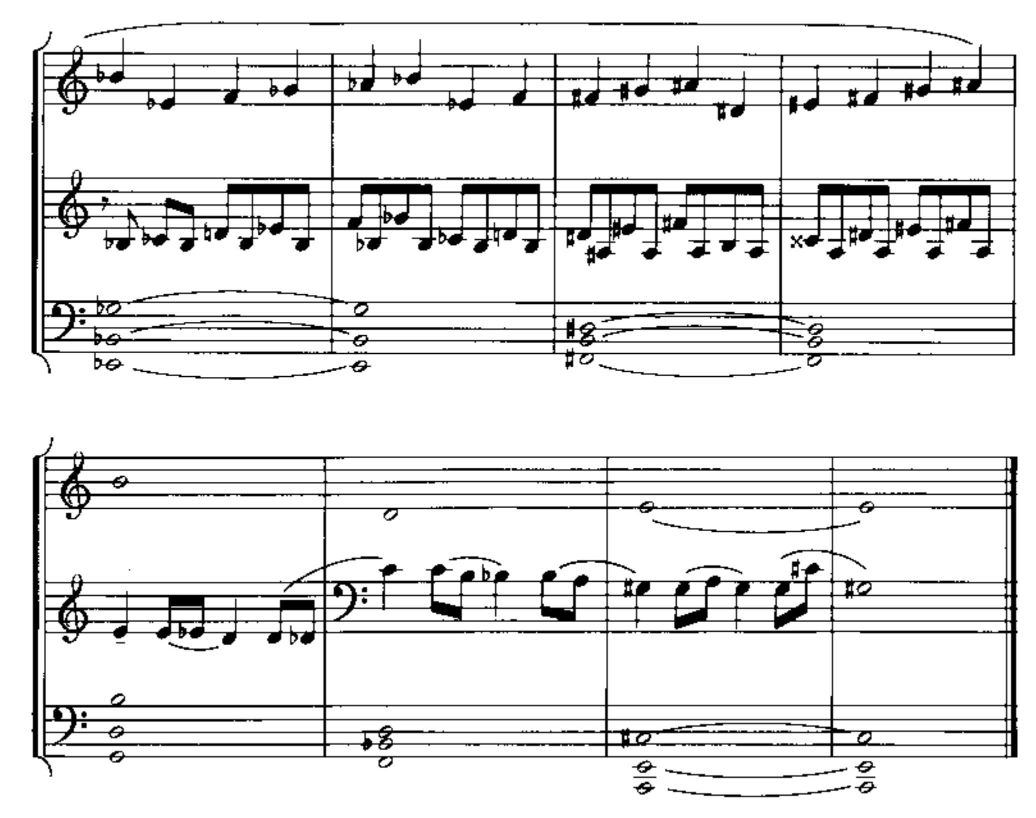


Fig. 12-12. Continued.

target that explicitly contains all the unique pitches in P and TM combined.

My "problem" was caused by forgetting and ignoring such details of the selection process. In using **FCHNG** for the sixteenth chord (I = 15) in the session just described, I assumed the selected structures would follow my voice-leading specifications *and* account for the single pitch of the target melody. Because the odds are quite good that an acceptable result will be achieved (the tension will be in the conventionally acceptable range), it was some time before a glaring breach in my lucky streak demanded that I review my logic. When the target melody contains only one pitch, that pitch will be in the chord unless the P vector has been reset explicitly or by an ancillary function. The composer must remember that certain functions do reset P and he must make use of tension limits at C-time to sort out unacceptable results. Just by typing **CNT**, he may be able to see if the target melody has been hit, although that will not tell him whether a miss has caused an unacceptable tension value.

Results of the Session

The remaining steps are far more enjoyable and rewarding than those followed in the terminal session. As before, the melody is transcribed into pitch notation by marking off measures according to the attack vector and assigning durations (see the top staff in Figure 12-12.) The accompanying chords are written out note-for-note, but it is best to "voice" them. (This process will be described later.) The bottom staff in the figure shows one of the possible voicings for the calculated harmony. To show where a composer might go from here, I include a suggestion for an accompanying voice in the middle staff.

Table 12-1. The MT Structures of Table 11-4, Compressed Into Scales.

							S	C	A	L	E	S		CLASS
2	1	2	1	2	2	:	C	D	D↑	F	F↑	G↑	A↑	-MI7
2	1	2	1	3	1	:	C	D	D↑	F	F↑	A	A↑	-MI7
2	1	2	1	2	3	:	C	D	D↑	F	F↑	G↑	B	-MA7
2	1	2	1	3	2	:	C	D	D↑	F	F↑	A	B	-MA7
2	1	2	1	4	1	:	C	D	D↑	F	F↑	A↑	B	-MA7
2	1	2	2	2	1	:	C	D	D↑	F	G	A	A↑	MI7
2	1	3	1	2	1	:	C	D	D↑	F↑	G	A	A↑	MI7
2	1	2	2	2	2	:	C	D	D↑	F	G	A	B	MI
2	1	2	2	3	1	:	C	D	D↑	F	G	A↑	B	MI
2	1	3	1	2	2	:	C	D	D↑	F↑	G	A	B	MI
2	1	3	1	3	1	:	C	D	D↑	F↑	G	A↑	B	MI
2	1	2	3	2	1	:	C	D	D↑	F	G↑	A↑	B	+MI
2	1	3	2	2	1	:	C	D	D↑	F↑	G↑	A↑	B	+MI
2	1	4	1	2	1	:	C	D	D↑	G	G↑	A↑	B	+MI
1	3	2	1	2	1	:	C	C↑	E	F↑	G	A	A↑	LG7
2	2	2	1	2	1	:	C	D	E	F↑	G	A	A↑	LG7
3	1	2	1	2	1	:	C	D↑	E	F↑	G	A	A↑	LG7
2	2	2	1	2	2	:	C	D	E	F↑	G	A	B	MA7
2	2	2	1	3	1	:	C	D	E	F↑	G	A↑	B	MA7
3	1	2	1	1	3	:	C	D↑	E	F↑	G	G↑	B	MA7
3	1	2	1	2	2	:	C	D↑	E	F↑	G	A	B	MA7
3	1	2	1	3	1	:	C	D↑	E	F↑	G	A↑	B	MA7
1	3	2	2	¯1	3	:	C	C↑	E	F↑	G↑	G	A↑	+7
2	2	2	2	¯1	3	:	C	D	E	F↑	G↑	G	A↑	+7
3	1	2	2	¯1	3	:	C	D↑	E	F↑	G↑	G	A↑	+7
2	2	2	2	2	1	:	C	D	E	F↑	G↑	A↑	B	+MA7
2	2	3	1	2	1	:	C	D	E	G	G↑	A↑	B	+MA7
3	1	2	2	2	1	:	C	D↑	E	F↑	G↑	A↑	B	+MA7
3	1	3	1	2	1	:	C	D↑	E	G	G↑	A↑	B	+MA7
1	4	¯1	3	2	1	:	C	C↑	F	E	G	A	A↑	7+3
2	3	¯1	3	2	1	:	C	D	F	E	G	A	A↑	7+3
3	2	¯1	3	2	1	:	C	D↑	F	E	G	A	A↑	7+3
2	3	¯1	3	2	2	:	C	D	F	E	G	A	B	MA7+3
2	3	2	1	2	1	:	C	D	F	G	G↑	A↑	B	+MA7+3
3	2	2	1	2	1	:	C	D↑	F	G	G↑	A↑	B	+MA7+3
4	1	2	1	2	1	:	C	E	F	G	G↑	A↑	B	+MA7+3

Listing 12-1. Listing of a session using HOFM and preference arrays to harmonize a melody produced by 4 DVLP2 2 7.

```
      PRF1
 0  1  5
 0  1  6
 1  0 ¯1
 1  0  5
      PRF2
 0  1 ¯1
 0  0  5
 1  0 ¯4
 1  0  5
 1  1 ¯3

      SA
 4 3
 3 4

      M
4 6 4 11 4 4 6 4 11 4 5 6 7 9 7 14 7 7 9 7 14 7 8 9 10 11 10 15 10
 10 11 10 13 11 7 8 10 6 5 6 8 10 3 5 6 8 10 3 5 6 8 10 11 2 4

      ATK←2 4 4 2 2 4 4 1 1 2 4 4 2 8 8 1 1 1

      PREF←PRF1



ATK HOFM M

I=0 ?

TM=4 6
S, P, XP, F OR XF (OR B) ?
S←F←1

CND=Γ/CNT?  (CND,CNT,HA,H?)

I=1 ?

TM=4 11 4 4
S, P, XP, F OR XF (OR B) ?

CND=Γ/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
1 STRUCTURES.   NOW CND=L/CNT ?

I=2 ?


TM=6 4 11 4
```

```
S, P, XP, F OR XF (OR D) ?

CND=F/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
1 STRUCTURES.    NOW CND=L/CNT ?

I=3 ?

TM=5 6
S, P, XP, F OR XF (OR D) ?

CND=F/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
2 STRUCTURES.    NOW CND=L/CNT ?

I=4 ?

TM=7 9
S, P, XP, F OR XF (OR D) ?

CND=F/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
1 STRUCTURES.    NOW CND=L/CNT ?

I=5 ?


TM=7 2 7 7
S, P, XP, F OR XF (OR D) ?

CND=F/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
1 STRUCTURES.    NOW CND=L/CNT ?

I=6 ?

TM=9 7 2 7
S, P, XP, F OR XF (OR D) ?

CND=F/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
1 STRUCTURES.    NOW CND=L/CNT ?

I=7 ?

TM=8
S, P, XP, F OR XF (OR D) ?
D
   1  1  4  8
   0  0  4  7
   1  6  9  1
   0 11  3  6
   1  4  7 11
   0  3  7 10
   1  9  0  4
```

```
CYCLE 3

CND=Γ/CNT?  (CND,CNT,HA,H?)
HA
   0  6 10  1
   1  6  9  1
CNT
1 1
0 TCNT 1000
2 STRUCTURES.   NOW CND=L/CNT ?
CNT
11 110
CND←Γ/CNT

I=8 ?

TM=9
S, P, XP, F OR XF (OR D) ?
0 FCHNG 2

CND=Γ/CNT?  (CND,CNT,HA,H?)
HA
   0 11  3  6
   1 11  2  6

I=9 ?
PREF←PRF2

TM=10 11
S, P, XP, F OR XF (OR D) ?

CND=Γ/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
1 STRUCTURES.   NOW CND=L/CNT ?

I=10 ?

TM=10 3 10 10
S, P, XP, F OR XF (OR D) ?

CND=Γ/CNT?  (CND,CNT,HA,H?)
1000 TPRF 2000
1 STRUCTURES.   NOW CND=L/CNT ?
CNT
1110

I=11 ?

TM=11 10 1 11
S, P, XP, F OR XF (OR D) ?

CND=Γ/CNT?  (CND,CNT,HA,H?)
100 TPRF 1000
1 STRUCTURES.   NOW CND=L/CNT ?
```

```
I=12 ?

TM=7 8
S, P, XP, F OR XF (OR D) ?

CND=Γ/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
3 STRUCTURES.    NOW CND=L/CNT ?
CNT
100 110 110

I=13 ?

TM=10 3 5 6 8 10 3 5
S, P, XP, F OR XF (OR D) ?

CND=Γ/CNT?  (CND,CNT,HA,H?)
0 TPRF 1000
1 STRUCTURES.    NOW CND=L/CNT ?
CNT
121

I=14 ?

TM=6 8 10 3 5 6 8 10
S, P, XP, F OR XF (OR D) ?

CND=Γ/CNT?  (CND,CNT,HA,H?)
100 TPRF 1000
1 STRUCTURES.    NOW CND=L/CNT ?

I=15 ?

TM=11
S, P, XP, F OR XF (OR D) ?
0 FCHNG 1

CND=Γ/CNT?  (CND,CNT,HA,H?)

I=16 ?

TM=2
S, P, XP, F OR XF (OR D) ?
1 FDOWN 0

CND=Γ/CNT?  (CND,CNT,HA,H?)
ρHA
4 4
0 TCNT 100
2 STRUCTURES.    NOW CND=L/CNT ?
CNT
11 0
HA
  1  9  0  4
```

```
 0 10  2  5

I=17 ?

TM=4
S, P, XP, F OR XF (OR 0) ?
S←0
F←2

CND=T/CNT?  (CND,CNT,HA,H?)

OK?

M:   0:   E       F↑
H:        C↑      E       G↑

M:   0:   E      B      E      E
H:        C       E       G

M:   0:   F↑      E       B       E
H:        F↑      A       C↑

M:   0:   F       F↑
H:        B       D↑      F↑

M:   0:   G      A
H:        E       G       B

M:   0:   G  +1D  ¯1G     G
H:        D↑      G       A↑

M:   0:   A      G  +1D  ¯1G
H:        A       C       E

M:   0:   G↑
H:        F↑      A       C↑

M:   0:   A
H:        B       D↑      F↑

M:   0:   A↑      B
H:        E       G↑      B

M:   0:   A↑  +1D↑  ¯1A↑     A↑
H:        A       C↑      E

M:   0:   B       A↑  +1C↑  ¯1B
H:        G↑      B       D↑

M:   0:   G       G↑
H:        E       G↑      B

M:   0:   A↑      D↑      F       F↑      G↑      A↑      D↑      F
H:        D↑      F↑      A↑
```

```
M:   0:   F↑   G↑   A↑   D↑   F    F↑   G↑   A↑
H:        B    D↑   F↑

M:   0:   B
H:        G    B    D

M:   0:   D
H:        A↑   D    F

M:   0:   E
H:        A    C↑   E

TO KEEP H, ENTER: 'NAME←H', OTHERWISE CR
```

13 Harmonic Continuity

Why do some chords sound more pleasant than others? In pondering that question, we had to employ reasoning that took us into territory normally thought to lie outside the realm of music. While we can't really answer questions that begin with "Why . . . ?" a good imitation of an answer—a hypothesis—will let us answer questions that begin with "How . . . ?" A hypothesis based on the abstract idea of order led to a theory of tension that allowed us to answer the question, "How can we rank chords according to their degree of pleasantness?"

Now we face an even tougher question: In harmonizing a given melody, why does one sequence of pleasant chords sound more acceptable than another sequence of equally pleasant chords? That is, in terms of our theory of tension, every chord in a sequence, in combination with the melodic segment it harmonizes, may have just the right subjective measure of tension. Yet the process in which we sense one chord "becoming" another as the sequence moves through time imposes an additional set of criteria for judging acceptability. Some sequences of perfectly acceptable chords may be unacceptable as sequences! Why? Again, we can't really expect to answer that question, but we can hope to form a hypothesis that can be tested by answering the more practical question, "How can we predict and calculate acceptable sequences of chords?"

The sensibility of individual chords, according to our assumptions, depends on the way the human mind perceives order in pitch space. It was not terribly hard to clear a logical path from the psychological perception of the octave to the idea of symmetry in a cyclically repetetive dimension (represented by a clock face), and from there, through the statistics of order, to a quantitative theory of tension. History then served as one of the tests for our logic. We were led to expect a correlation between tension and chronology, and, in fact, found that the higher the tension in a class of structures, the later it was introduced into Western music.

As a consistent extension of our hypothesis, the question of acceptable continuity must depend on the way our senses integrate time into the sensibility equation. Perhaps it is appropriate that when time is included we need to turn directly to history, not as a test, but for our initial inspiration. No recognizable forms of order or symmetry leap to mind, so our immediate goal is to examine trends in time in order to see if we can discern tendencies that suggest what constitutes order in the temporal continuity of pitch in Western music.

PITCH AND "TONALITY"

It seems evident that our short-term musical memory functions in a way analogous to a long-persistent phosphor on a video display; successive notes of a melody paint at least part of a coherent picture in pitch space before they fade away. As Western melody evolved in terms of quantized pitches, before one person's melody could be remembered and appreciated by another—and certainly before that melody could be sung by someone other than the composer—agreement had to be reached on matters of intonation. Only with the technological development of instruments based on vibrating strings or linear columns of air could tuning systems and discrete scales of pitch become objective. This development also introduced the word *harmonic* into music and mathematics. It was found that two strings with identical properties, but with lengths in the ratio of some entry in the *harmonic series* (1/1, 1/2, 1/3, 1/4, 1/5, . . .), exhibit a noticeable lack of interference in their vibrations so that, when plucked simultaneously, they emit a "harmonious" sound.

In fact, it turns out that the vibrations of a single string or air column can be described in terms of combinations of harmonic frequencies. The relative amplitudes of the various harmonics determine the timbre or tone color of the sound and are determined by the various physical properties of the instrument. Normally, one hears the fundamental frequency as "the" pitch and the higher frequency harmonics, having much smaller amplitudes and vibrating with the fundamental in a "coherent" way, are hard to observe directly as tones. Our senses tend to interpret the harmonics not as notes at all but as an entirely different component of sound—they give a certain quality or color to the fundamental pitch. If particular harmonics are loud enough to be heard as tones, or if the instrument has nonlinear modes of vibration (plates, bells, taut membranes, etc.), the combined audible pitches produce an effect ranging from tone color to tonality.

The first three octaves of the harmonic series, with C as the fundamental, contain *approximately* the pitches shown in the top staff of Fig. 13-1. Note the tonality here for the first seven unique tones (up to the thirteenth harmonic) is the same as that for the second structure in the LG7 class in Table 11-4.
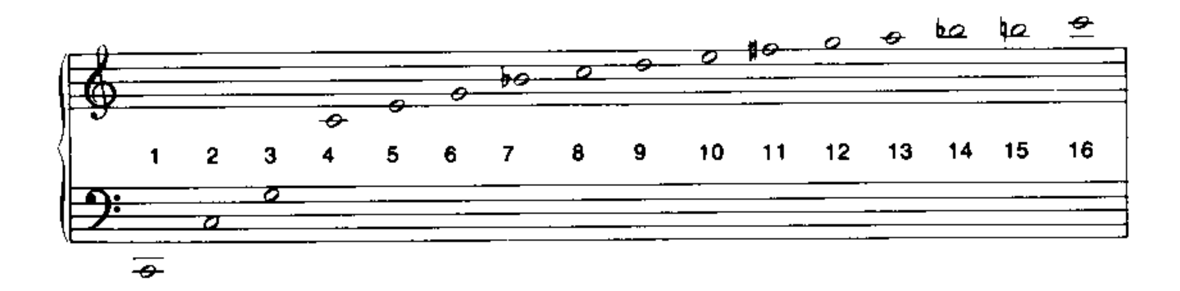
SCALES AND TONALITY

The development of scales of well-defined tones had to move at a glacial pace because each tone carries its own tuning system as well as its own tonality. That is, if we tune the pitch D in a way consistent with its appearance as the ninth harmonic in the figure, the harmonic series built on D becomes the one shown on the bottom staff. Notice the seventh harmonic of this fundamental introduces the pitch C, with a relative frequency of 63/8 instead of 64/8 or 8! Similarly, harmonics 9, 10, 12, 13, and 14 introduce

E	at	81/8	instead of	80/8	or	10
F♯	at	45/4	instead of	44/4	or	11
A	at	27/2	instead of	26/2	or	13
B	at	117/8	instead of	120/8	or	15
C	at	63/4	instead of	64/4	or	16

Western ears were able to overcome such subtle problems and concentrate on finding (and arguing about) the "correct" way to tune scales of "fundamental" tones.

The development of scales with their own tonality required testing and proving techniques of sufficient interest to entertain the intellects of com-



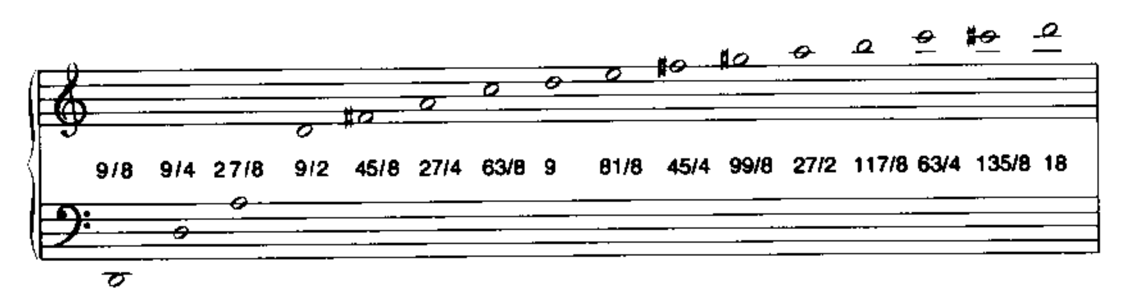
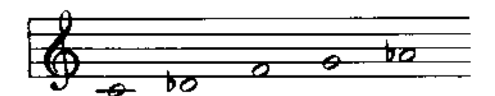


Fig. 13-1. The harmonic series built on C is shown on the top staff. The bottom staff shows the harmonics of D tuned according to the ninth harmonic of C and expressed as frequency ratios with respect to C.

posers, singers, and philosophers for many generations. Such techniques were found using different *modes* of the same scale. An example of a "mode" can be experienced by playing the scale of white notes starting on a pitch other than C. Or, even better, play a simple (all white-note) melody but shift all the tones consistently so that the starting point is different than in the original version. Modes provide a glimpse into the subtle workings of tonality as a driving force in music.

Notice, we're talking about monophonic music here—music consisting of a single voice. Earlier, I implied that a "complete" scale carries a tonality. Permuting such a scale in various ways raises some disturbing questions. Consider just the "incomplete" pentatonic scale:



Notice the difference between modal settings of the "same" melody in that scale (Fig. 13-2). Though monophonic, each modal setting carries a different "virtual" harmonization to the listener's mind.

This immediately raises questions about the meaning of tonality with respect to "a" scale and may help clarify what I mean by a "complete" scale. If we consider a scale to be a collection of tones independent of its starting point, it appears that its "complete" tonality can contain "subtonalities" that suggest other complete tonalities. That is to say, a particular melodic segment (containing fewer than seven unique pitches) might be contained in different complete scales. Different listeners might then hear different tonalities suggested. It may even be that most listeners in our culture will hear one particular tonality, the most likely one being the structure whose first three or four tones (in the harmonic or expanded form of the scale) score highest in harmonizing this target melody.
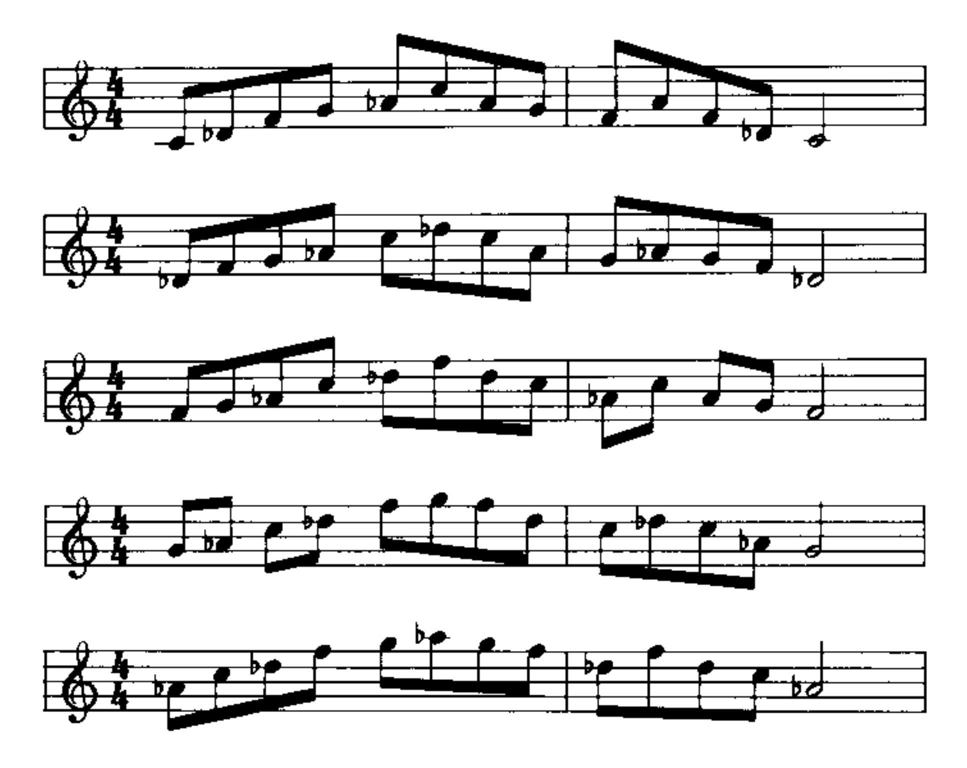


Fig. 13-2. The modal settings of a melody in the pentatonic scale shown in the text.

THE GROWING SENSE OF HARMONY AND TONALITY

Over a period of about 800 years beginning in the ninth century A.D., polyphonic music developed. The earliest use of multiple voices relied on the technique of *coupling* or duplicating the melody, note for note, with the voices separated by a "perfect" interval (octave, fourth, or fifth). Always thinking melodically, composers avoided the onset of boredom, making the aural diet richer and more spicy by allowing increasing freedom to the individual voices. Combinations of two, three, or four contrapuntal melodies, each doing its own thing, had to initiate a conscious awareness of harmonic forces.

The beginnings of explicitly harmonic music came around the seventeenth century. It is no coincidence that this was a time of revolutionary technological advance. Existing musical instruments were so greatly improved that, for the first time, instrumental music began to rival vocal music in the historians' scale of importance. It was now possible, and more necessary than ever, to settle on objective forms of intonation. In terms of today's piano keyboard, the white notes were tuned according to various logical arguments based on the ratios of the frequencies of the different intervals in the harmonic series. When it came to tuning the black notes, there were too many conflicting logical arguments for a consensus to be reached in settling the matter.

In tuning the white notes, the fourth and seventh scale steps (F and B) were particularly troublesome. Notice that they are not even in the tonality of the first seven unique notes in the harmonic series for the fundamental, C. The nearest black notes—F♯ and B♭—are in this part of the series, but their tunings were also vexatious. They became the center of attention when composers noticed that by substituting either one of them for the corresponding entry in the scale of C—F♯ for F or B♭ for B—a marvelous change of tonality took

place. Altering the seventh step to a "reasonable" B♭ made it the fourth step in the scale of F. Changing the fourth scale step to approximate F♯ made that the seventh scale step in the key of G. Further alterations of the fourth or seventh from the keys of F or G soon brought on conflicting claims for the proper tunings of the remaining black notes. Besides, even the F and G scales had slightly different characters from each other and from the scale of C because of the differences between the tunings in their relative scalewise intervals.

But the sequence of events now offers a vital clue for our hypothesis. First, after something resembling a scale was established, there came modal music—one complete tonality with varying apparent root tones. Then came the single alteration that changed the complete tonality to that of a scale whose root is up or down a perfect fourth. Of course, a "complete" tonality, if such a thing really exists, could only have been subliminal when working in an incomplete tuning system. But now mathematical thought enters the picture to accompany the technology.

Logarithms had been invented by John Napier, and over a period of roughly a century an inevitable though somewhat abstract idea percolated. Why not drop the current tuning system in favor of one in which all the intervals are tempered in a logarithmically equal fashion? Johann Sebastian Bach embraced the idea with enthusiasm, and musical thought now turned to take advantage of and explore the new equality between "tonalities."

HARMONIC VERSUS MELODIC TONALITY

A distinction between harmonic and melodic thought is not at all easy to observe, but certain subtle differences do emerge when we apply our ideas on tension. The melodic scales we have inherited still contain those harmonically extraneous fourth and seventh steps. When we look at the tension of each note in a major scale with respect to a major triad built on the root tone of that scale, we see the values

0 11 0 1001 0 10 100.

It would require a startlingly stannous ear to equate the harmonic effect of the C-major scale with a C-major triad.

By "reharmonizing" a major scale with other major triads, the resulting tensions are revealing. In particular, if we harmonize the same melodic scale with a triad built on the fourth scale step, we find:

0 10 100 0 11 0 110,

and, if we reorder the scale so that it begins on that fourth step, the tones now display their tension in the order:

0 11 0 110 0 10 100.

Notice that the fourth and seventh steps are still "accented" here, but the important thing to realize is that a C-major scale makes more harmonic sense when related to an F-major chord; a C-major melodic tonality has an F-major harmonicity or harmonic tonality.

Examination of the tensions of the same notes with respect to a major triad built on the fifth scale step shows

1001 0 10 1 0 11 0,

or, rearranged with the harmonic root first:

0 11 0 1001 0 10 1,

accenting only the fourth "scale" step. This shows that a melodic segment extracted from a C-major scale can be harmonized with a notable lack of tension by a G-major structure as long as the pitch C is avoided in the melody.

Although the above relationships were recognized without any reference to numerical values of tension, the idea of a "complete" structure applied only to melodic scales. Harmonic structure was approached rather schizophrenically: A chord could be built (in thirds) on any note in a melodic scale but, regardless of its root tone, the notes of the chord were referred to as first (tonic),

third, fifth, etc., as if they came from a different scale. Yet the chords themselves (not the notes) were also numbered according to the position of their root in the original scale—I, II, III, etc.—and a complex set of rules was worked out empirically that described what chordal sequences were "sensible." Thus, while a melody existed in one scale, it could be harmonized, according to the rules, by any of the "incomplete" tonalities (chords) contained in that scale. In practice, a composer could modulate to another key through the use of a *pivotal chord*—one whose tones are common to the current scale as well as to the new one. From that point on, melody and harmony could behave according to the rules as though they were in the new key. These rules and the ways they were extended to allow modulation to various keys encompass the logic our hypothesis must imitate.

COMPLETE (HARMONIC) TONALITIES

Our next assumption will be that harmonic structures, like melodic scales, can be limited to seven notes. Other forms of "completeness" seem equally reasonable considering that the four most symmetric triads can harmonize nine of the twelve pitches in our tuning system with tensions below 1000, and six of these same pitches have tension less than 100. In fact, there are occasions when a melodic segment consists of consecutive semitones (not in any conventional melodic scale) and nine-note harmonic structures would make harmonization easier. However, in settling for seven-note structures, we'll find it easier to grapple with the idea of order in continuity in a useful and self-consistent way.

Traditionally, the seven pitches in any scale must use the seven letters from A through G with accidentals (sharps or flats) as required. Because the letters are taken in sequence, the various arrangements of accidentals that are allowed prevent notes from duplicating or crossing each other, and so give rise to the property of melodic scales mentioned earlier—viz., scales are composed of whole-tone and half-tone intervals in roughly a two-to-one ratio.

In forming harmonic structures as though they were expanded scales, accidentals can be assigned more freely without causing letters to cross each other in the expanded sequence (A, C, E, G, B, D, F) or duplicating other notes in the structure. The 36 structures shown in Table 11-4, despite an apparent break with tradition, are all built in "thirds," as tradition requires. They all begin on C and the following six tones are variants of E, G, B, D, F, and A, in that order. The apparent discrepancy arises because my algorithm refuses to recognize different spellings for the same tone. Thus, D¦ can mean D♯ or E♭, and the letter G might even stand for A♭♭ (double flat) or F♯♯ (double sharp) as well as G-natural. But when the structure is compressed to form a "scale," we may have to rearrange the notes in that scale to maintain alphabetic order (see the scales in the classes labelled "7 + 3" and "MA7 + 3" in Table 12-1).

THE CLASSES OF TONALITY

Our list of 36 structures is longer than that used by individual composers. It grew out of a list of only a dozen four-note chords suitable for MT. Each structure has one of these tetrads for its four lowest tones, and all structures built on the same tetrad comprise a class. For example, all the major-seventh (MA7) structures begin with C, E, G, B, and all the minor-seventh (MI7) structures have C, D¦, G, A¦ as the first four pitches. The number of structures in a class depends on the number of ways the three additional letters (D, F, and A) can be added above the tetrad with appropriate accidentals to produce a tonality whose total tension is below 1000, or else tonality with higher tension that has achieved acceptability through conventional use.

The practical distinction between class and tonality may be hard to discern for those with less-experienced ears. The problem lies with the increase in tension contributed by the three highest tones in the structures. As suggested earlier, discrimination between elements in a complex class requires enough familiarity with the material to recognize individual patterns and their differences;

tonalities that sound equally strange can't be identified individually. Further, much composition is driven by harmonic tension rather than by melodic tonality, so that we become accustomed to hearing tetrads melodized by various combinations of nine low-tension tones without regard for a specific melodic scale. Naturally then, a chord with fewer than seven tones may seem to carry all the harmonic information that is really necessary and can harmonize a melodic line that freely mixes tonalities in the same class.

Still, to place our hypothesis on a firm foundation, I'm going to assume that discrete seven-note tonalities impress themselves on our senses, at least subconsciously. It may be that, once our minds zero in on the "correct" tonality for sensible continuity, some degree of "relaxation" sets in, allowing us to accept pitches from other members of the same tonal class.

CONFLICTING TONALITIES AND KEYS

Before the details of the remainder of the hypothesis are worked out, there remains one aspect of tonality that I should try to clarify. It concerns the fact that different structures may contain exactly the same pitches, though in a different order. The conflict between a C-major melodic tonality and an F-major-seventh harmonic tonality (from the MA7 class) was introduced above. Ordinarily a listener actually will identify the key or tonality of a melodic segment by its harmony; a C-major melody harmonized by an F-major chord will seem to be the key of F. Remember, both of these tonalities contain the same notes (all the white ones):

C D E F G A B

for the (C-major) melodic scale and

F A C E G B D

for the harmonic structure. But there can also be redundancy between two or three harmonic tonalities. For example, one of the D-minor seventh (MI7) structures also contains just the seven white notes

D F A C E G B

thereby duplicating the pitches in the F-major seventh tonality above.

Like a connoisseur of wines, one must have adequate experience to taste the difference in the flavor of such tonalities. To aggravate the problem, a tetrad called a "sixth" chord was introduced into common use, before the major-seventh tetrad was, to convey a major tonality. An F-major seventh chord:

F A C E

has a tension of 100. Built on F, the sixth chord, with a tension of only ten, would be

F A C D,

and it too can have an F-major "flavor," even though it is really a D-minor seventh tetrad! (Built on D, the chord is D F A C.)

Taken as static chords, any argument over apparent tonality would be purely academic. But in continuity there are nuances that persuade us whether to accept D or F as the root in any particular situation. From our cultural experience, we expect melodies and their accompanying chords to have either major (MA7) or "true" minor (MI) tonalities at stable or resting points in the phrasing. In unstable or moving positions of the continuity, the melody can appear to be in transition between "keys," so other classes of harmony are quite acceptable. It appears then that harmony is not quite in complete control of our sense of key, and we should probably restrict the idea of "key" to imply the *sensed* melodic tonality regardless of the actual harmonic tonality. Melody must seem to move in a key or tonality, even though that key can change at any time. Chords instantaneously define a (harmonic) tonality that may, under influence of the melody, *suggest* a different but related melodic tonality or key.

In addition, certain classes, e.g., the "augmented minor" (+MI), are so alien to our experience that they evoke a more familiar class that has the same pitches redistributed in its tonalities. For example, one of the +MI structures, if built on the pitch, A, has the notes

A C E#(F) G# B D G

which are also present in the F-major-seventh tonalities

F A C E X B D,

where X can be either G or G#. When this structure harmonizes a melodic segment, a listener will almost certainly claim to hear an F-major tonality. Again, this feeling of key must be blamed on the melody. The listener will surely taste the difference between the class structures (tetrads)

A C F G# and F A C E

as well as between

F A C E and F A C D

though our connoisseur might smugly describe them in terms of relative crispness or bouquet. (Our aural and oral senses evolved so long before language that they bypass the verbal/conceptual links to our emotions, making such sensory stimuli and their subtle differences strictly ineffable.)

THE HYPOTHESIS

The historical trends suggest that we seek some quantitative way to connect order in continuity with the change in the complete structure that occurs on going from one tonality to the next. That is, the earliest harmonies were restricted to a single key, so we might suppose the highest degree of order occurs when consecutive tonalities have the same set of seven tones—as in the F-major and D-minor seventh structures shown earlier. We saw that the next breakthrough came, historically, with the change of a single tone to modulate to a new key. So let's say the lowest level of disorder is introduced when adjacent tonalities differ by a single pitch and have six tones in common. The historical argument can be made to take us one step nearer to the point where two tones change between successive tonalities, so let's look at the obvious extension.

There are only six categories into which pairs of tonalities can fall:

Category	Pitches changed	Pitches common
1	0	7
2	1	6
3	2	5
4	3	4
5	4	3
6	5	2

The number of changed pitches plus the number of common ones must equal the number of notes in a structure, seven; with only twelve unique pitches available, two structures can differ by at most by five pitches. Experience with similar situations in other disciplines suggests that we examine the statistics here more closely before jumping to any conclusions about the direction of increasing disorder among these categories.

If we take all 36 tonalities into account, a single pitch can be harmonized in any of $7 \times 36 = 252$ ways. Requiring more than one pitch to be harmonized by a single structure can decrease that number substantially. For statistical arguments to have any validity we need a large number of samples; if 252 is the most we can manage, we had better make full use of them to see whether they distribute themselves among our six categories in a way that makes sense statistically as well as musically. Remember from our arguments concerning the connection between disorder and statistics that chaos should be far easier to achieve through random selection than order. We'll look to the populations in the six categories for clues concerning relative amounts of order, assuming a category that has far fewer entries than other categories must appeal to our senses as being more ordered.

A distribution pattern among the six categories can be evaluated as follows. First, an initial tonality and a single note to be harmonized will be selected. Next, we'll construct the array of harmonic alternatives that harmonize the given tone. Then we will simply count the number of these structures that fall into each of the six categories by comparing each directly to the given initial tonality. For example, if our first tonality is an A♭-major structure corresponding to the eighteenth one in our list, and C is the pitch to be harmonized by all chords, then the six groups are found to be populated as follows:

No. of changes	0	1	2	33	4	5
No. of tonalities	4	26	82	99	39	2

This says that four structures contain exactly the same seven pitches as the initial **Ab-MA7** (including the initial structure itself). Twenty-six structures differ from the initial tonality by a single tone. Only two structures were found to have the maximum number of different pitches.

Changing the first chord to a D-minor seventh (the sixth one in the list) alters the distribution to:

No. of changes	0	1	2	3	4	5
No. of tonalities	4	26	74	93	46	9

Experimenting in this way, one finds the distributions all peak in the middle rather than at the right as you might have expected. That is, you may well reason that the more changes we allow, the greater the number of ways such changes can occur. But that would be true only if all seven pitches could change in many ways. Limiting those pitches to just 12 possible tones imposes a different type of order at each end of the distribution. At the left, we have tonalities with the maximum number of common pitches; on the right, we see tonalities having the maximum number of changed pitches. Conversely, on the left lies the minimum number of pitch changes, and on the right, the minimum number of common pitches. There are many physical systems having symmetries that make minimal change display one type of order and minimal constancy another. Between the limits, disorder must reach its peak.

If we were dealing with millions of possibilities rather than 252, we could expect quicker verification or rejection of this hypothesis on purely intellectual grounds. Our esthetic sensitivity would be more likely to distinguish between "ordered" and "chaotic" categories whose populations differed by at least a few orders of magnitude. Experimental verification also may be elusive because most of our 36 tonalities will not "strike a familiar chord" with many listeners. However, an approach has been found that suggests there is practicality if not truth in this hypothesis.

The entire list of 36 structures can be used to construct the harmonic alternatives for each melodic segment. Each structure then can be assigned to one of the six categories by counting the numbers of pitch changes required on going from the current tonality to this alternative new one. At this point we can select only the more familiar structures from each category for testing (in index origin zero, these would be rows 5, 7, 15, 17, and possibly a few others in our list). In doing so, we find individual and collective evaluations of sensibility in continuity do agree very nicely with the hypothetical predictions based on the full 36-chord complement. Trite chord changes are indeed found in the first category. Tonalities with one pitch change produce extremely common sequences. Less usual, but still familiar sequences are obtained by selecting from the far end of our spectrum where the maximum number of pitch changes occur. The intermediate categories—those with two, three, and four pitch alterations—sport "interesting," "unusual," and "uncertain" changes of harmony. Most of those labeled "unacceptable" are found in the group with three varied tones, and, if the comments of different observers with comparable musical experience can be trusted, the difference between "unusual" and "unacceptable" is subjective.

But our arguments and tests have involved sequences of only two successive chords, and a single chordal change can hardly be said to constitute harmonic continuity. A hypothesis that tries to take

this into account will seem more complete but be completely unverifiable. Suppose a particular sequence is to have just eight chords, an initial one followed by seven changes of tonality. Further, suppose each harmonization must account for only a single melodic pitch so that each chord, including the first, has 252 rivals for its place in the sequence. While each change of tonality can be analyzed by a distribution among the same six categories, the entire sequence of changes can be considered at a higher, more complex level of order.

First, there may be a relatively small group of eight chords, all of which contain the same seven pitches. (For this group to exist, every note of the melody must also be a member of every chord in the sequence.) At the other extreme would be another "small" group of eight chords that change maximally from each other. To get an idea of how large "small" is here, we might speculate that, on average, each of the 36 structures possible for any one harmonization could be followed by either of only two structures to fulfill the stringent re-
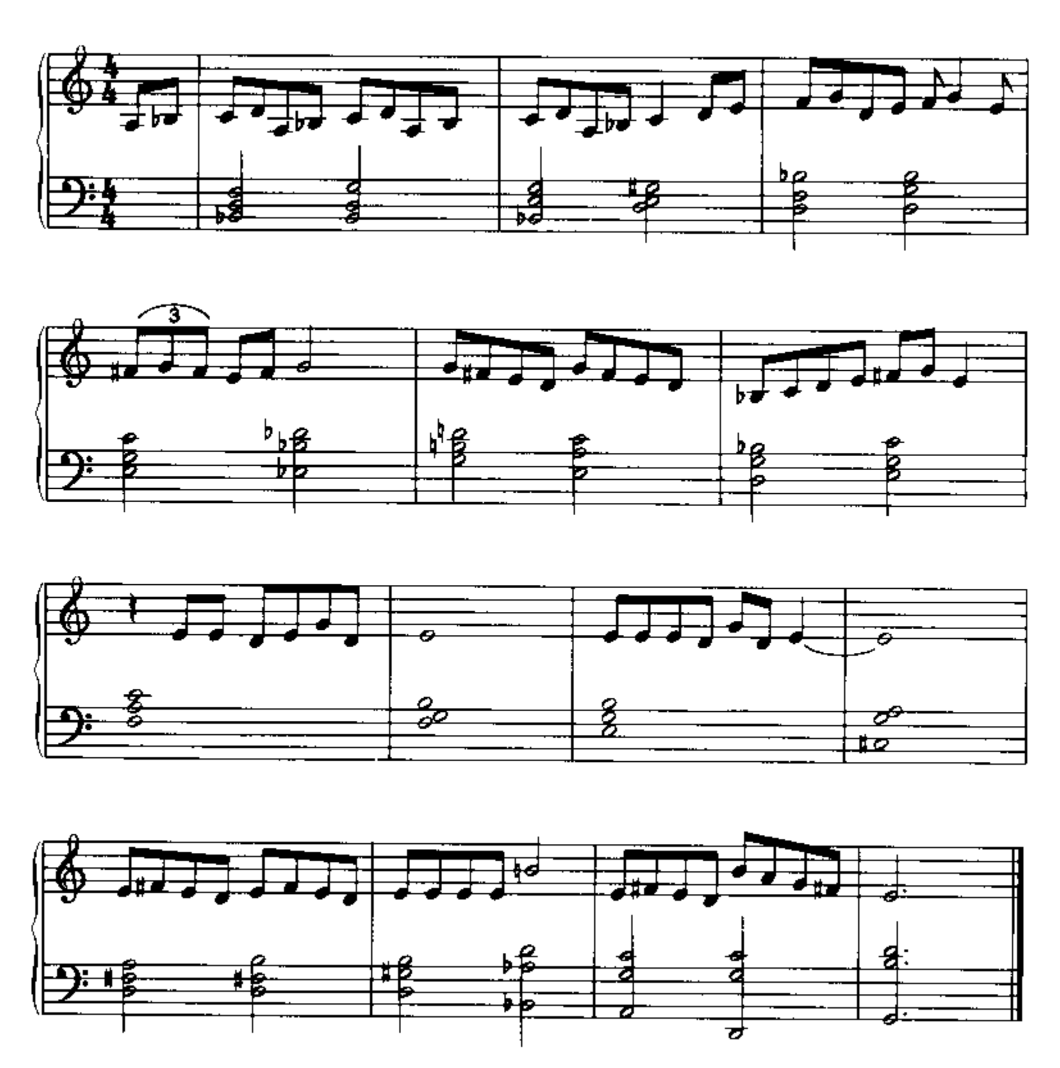


Fig. 13-3. A melody with harmonic continuity controlled by selecting chords from the lower populated categories of order (© 1978 Jaxitron).
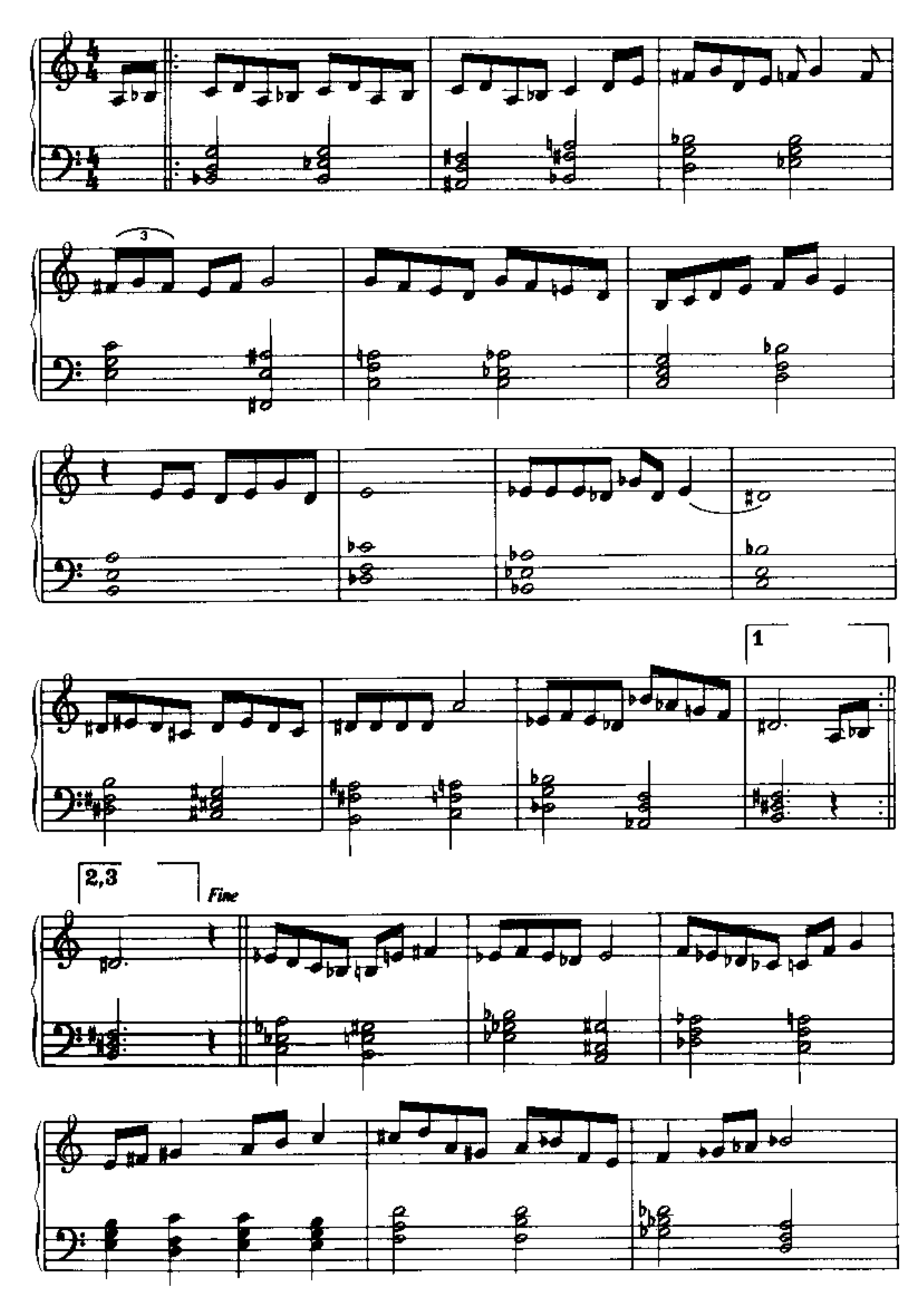


Fig. 13-4. Another treatment of the same melody, combining other levels of order and additional techniques (© 1978 Jaxitron).
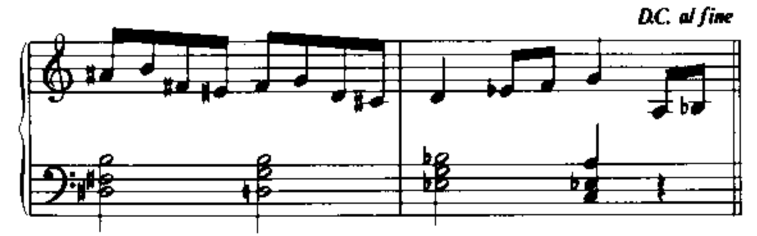


Fig. 13-4. Continued.

quirements of this category. This would present us with 252 × 2^7 possible eight-chord sequences, or "only" 32,256. This is a reasonable ballpark figure for both extreme categories, assuming the one with no changes exists at all.

The figures quickly become most unreasonable, however, when we try to tabulate the number of sequences that permit only a single pitch alteration to occur somewhere within the eight tonalities, or just two such changes, or three, or . . . . Notice that the maximum number of pitch changes would be five for each of the last seven tonalities, or 35 in all. This does not necessarily mean that our categories now run simply from zero pitch changes to 35. For example, "the" category with seven pitch changes might be broken into

1) Seven consecutive changes of one pitch
2) Five changes of a single pitch and one change of two pitches
3) Four changes of one pitch and one of three
4) Three single changes and two double
5) Two single changes, one double and one triple etc.

To appreciate the numeric magnitudes involved here, let's consider one highly "ordered" category. A commonly acceptable sequence might well be one that combines just one or two changes for each of the seven pairs of tonalities. If on the average there are 30 ways to have a single change of pitch and 80 ways to alter two pitches between successive structures, then the number of eight-chord sequences that have six single-note changes and one change of two notes would be 252 × 80 × 30^6 × 7 or over 100 trillion! (There are 252 ways to select the first chord; 80 ways to select a chord that changes two pitches; six chords, each of which has a single change and can be selected in 30 ways; and seven ways to permute the sequence of single- and double-tone alterations.) This refinement puts our hypothesis on a reasonable statistical basis but undermines any hope for testing or practical use.

Despite the apparent depth of the idea behind the statistical hypothesis, selecting a tonality according to the number of new notes it introduces is easy to program and, more important, to apply. A number of modifications have been made to the HOFM function to allow us to use this tool. Some of these were suggested directly by the "ordering" concept, others were "second-order" effects. The next chapter will show details of the approach with listings of the session that produced the examples shown here. The first (Fig. 13-3) exhibits the results of a fairly strict and unimaginative application of the statistical approach. The second (Fig. 13-4) uses the same approach but delves into the more "interesting" levels of order. Notice, the melody is not quite the same in the two examples, and the second one has been extended into the AABA form. Also note that, after using complete seven-part tonalities to determine the harmonic continuity, we present only three-note chords to the listener's ears, leaving his senses to detect as much melodic and harmonic tonality as is consistent with his musical experience.

14 A Systematic Approach

Composers—and artists in general—pay extravagant lip service to the idea of freedom. The sad fact is that our minds just don't function comfortably in a truly free (chaotic) environment. Whether listening to music or creating it, we need to see and enjoy finding the relationships that exist between entities within a highly structured environment. As disturbingly Machiavellian as it sounds, freedom requires order. Of course, the balance is delicate and probably even subjective. Programs such as the **HOFM** function for harmonizing melodies or **FORM** for constructing them contain a fair amount of logic, but perform a limited set of tasks—tasks which have been compared above to accounting or bookkeeping. They leave you, the composer, so free that, until you try doing without them, you may question whether they serve any purpose at all. In using such "driving" functions, you begin to develop a library of short, auxiliary functions (such as **TNSN**, **SCLIT**, **UNIQ**, and so on) that carry out useful operations. And you'll probably feel elated by the freedom to do so without recognizing that you really need to do so.

At first, the value of your library of auxiliary functions grows in direct proportion to its size. But before long, as the growth curve continues upward, the value curve starts to level off when the need for a "card catalog" becomes apparent. A time will come when you want to perform a particular operation, but you can't remember the name of the function that carries it out. Even worse, there will be times when you forget you have such a function, and you'll write a new one (with a new name!) to do the same job. Sometimes it will be convenient to see a list of the names of all your functions simply to serve as a menu to suggest what your next operation should be. Once you have the name of the function you need, there remain additional questions that may embarrass your memory: What left and right arguments are required? What global variables are needed? What globals will be reset? Will this function call any other functions and, if

so, can they affect any variables in the workspace?

TOWARD AN OPERATIONAL SYSTEM

In my harmonization/melodization workspace, the first voluntary curtailments of freedom took shape in establishing naming conventions for functions and variables and in trying to standardize the use and meaning of right and left arguments. Later, more freedom was given up as a satisfying style of using my functions evolved. That style began to dictate how functions should be designed to work in a cooperative way. The next step involved a conscious attempt to organize functions into categories and limit the number of functions that perform similar tasks. Roughly speaking, three categories of functions developed: those that affect the harmonic alternatives (HA array) those that operate on the target melody (TM vector and/or TGM array), and those that serve various utilitarian functions. Programs that cut across categorical boundaries were redesigned and rewritten to eliminate duplicate operations. The goal was to limit the way any particular operation could be performed without limiting the number of different things that could be done. Still, one thing that could not be eliminated was the need for that "card catalog." In fact, it soon became apparent that something more in the nature of a reference manual was needed!

The appendix in this book contains such a manual. It is fairly complete and self-explanatory. Unfortunately, as is too common with such documents, the information was packaged by and for the developer rather than for a potential user. However,

HA Operations

Function	Time	Use
F ALL P	C	keep HA's for which all HA[;FN F] ∊ P or all P ∊ HA[:FN F]
CLASS N	C,S	keep HA's for which CLS ∊ N
N EQH I	C,S	force HA's = H[I;] transposed through N
A F2FA B	C	keep HA's in which all H[I-1;FN A] ∊ HA[;FN B] or all HA[;FN B] ∊ H[I - 1;FN A]
A F2FN B	C	keep HA's in which no H[I-1;FN A] ∊ HA[;FN B]
A F2FS B	C	keep HA'S in which some H[I-1;FN A] ∊ HA[;FN B]
A KEPS B	C	keep HA'S for which corresponding A ∊ B where A can be TNS, FRD, CLS, etc.
A KRNG B	C	keep HA'S for which corresponding A is between 1↑B and −1↑B
NARROW N	C	keep only HA[N;] (& set all related entries - TNS, CLS, etc.)
F NO P	C	keep HA's in which no HA[;FN N] ∊ P
CY OVRIDC CL	C,S	HA forced to these cycles & classes regardless of TM (may not fit TM!)
CY OVRIDS STR	C,S	HA forced to these cycles & structures regardless of TM (may not fit TM!)
PRTL N	C	TNS of TM reset against HA[;FN N] and CND←└/TNS
M PRTLTN N	C	TNS of M set against HA[;FN N] and CND←└/TNS & new TNS displayed
F SOME P	C	keep HA's in which some HA[;FN F] ∊ P
STRUC N	C,S	keep HA's with these structures = i.e. HA[;0] ∊ N

Fig. 14-1. Reference sheet for the auxiliary functions that affect the Harmonic Alternatives (HA).

TM Operations

Function	Time	Use
ADJUST		used automatically at end of S-time and C-time to update TGM if not equal to TM
R CHNGTM V	C,S	TGM[R;] transposed by V — same V used on all R(ows)
S CNKTM N	C,S	performs octave adjustment of all notes in current TM so that all intervals in (N ≥ half an octave)
N EQM K	C,S	TM reset to TGM[K;] transposed through N S, TM are < N. (S is starting note—usually last pitch of previous TM)
FITM F	C	each TM entry set to nearest HA[BEST;F] where BEST implies the closest fit to CND
FITMALL F	C	all TM entries set simultaneously avoiding duplicated tones - i.e. F must have at least as many unique elements as does TM
I IROWTGM V	any	sets V in TGM[I;]
IC MLDZ IS	C	TM (re)defined by scale-interval sequence in IS using last pitch of previous TM as starting point and best HA as scale. The starting pitch will be the first one that works in the set: (¯1 ↑ TGM[I-1;] + chromatic intervals in IC) closest pitch in HA to last pitch
RESETM I	any	TGM rows from I[0] to I[1] reset to original values
UPTM V	C,S	V placed in TM and TGM[I;]

Fig. 14-2. Reference sheet for the auxiliary functions affecting the Target Melody (TM).

its order and compactness should be beneficial after demonstrating some terminal sessions. In using the workspace, I prefer to keep handy a highly adumbrated "crib sheet" (Fig. 14-1, 14-2, and 14-3) containing information on each of the three function categories crammed onto a single page. This will be of value only when the user has gained experience with the "system" and familiarity with the variables. Instead of presenting the many details in a formal way, I'll introduce them as they occur in sample terminal sessions.

To compose with the system, the user decides first on a "musical" operation. Then he must decide which of the available functions to apply and how they should be called. Whenever a desired operation simply can't be done, the system must be enlarged. As the system grows, it incorporates more elements of the user's style so that such additions become rarer. Because *style* is the keyword here, and I know there are elements of my own style that still elude my conscious observations, I can't pretend to define what a complete system should contain. Although I haven't bothered to change the name of the function, **HOFM**, a revised version of it is the "driving" function for the basic compositional system to be demonstrated here. I say the system is "basic" because it includes melodization as well as harmonization, but does not contain the more advanced functions needed to compose contrapuntal melodies and harmonies,

voicing, or orchestrative style.

THEMATIC MATERIAL FOR THE SESSIONS

The sessions about to be described produced the music shown at the end of the last chapter (Fig. 13-5 and 13-6). The thematic material for those pieces was worked out in a previous session using the DVLP routine presented in Fig. 14-4.

This function, like the previously demonstrated **DVLP2**, was an "architectural" exercise rather than a "formal" one. One of the simpler tunes produced through its use was created by the statement

```
M ← ⁻3 + 1 1 1 DVLP 1 2 2 2 2
```

The right argument sets up the kernel for a scale. The left argument (1 1 1) establishes a featured motif consisting of three successive unit intervals (four successive *scale* steps). The minus three, preceding the whole, adjusts the result to a given vocal range. In Fig. 14-5, typing the name, M, caused the DVLPed melody to be displayed. The clean listing hides all evidence of the counting process that led to the next statement, establishing the chordal rhythm in terms of melodic attacks. The

Info and Utility Operations

Function	Time	Use
BACK N	any	returns to beginning of: current time if N = 0 previous time if N = 1 I-time if N = 2
DSPLAH		to see H array when execution is stopped
FN F	C,S	converts harmonic functions (1, 3, 5, . . ., 13) to HA indices (1-7) (0 is structure index)
FRDGRP	C	displays FRD distribution of HA
HPATTERN		constructs ATK vector with a dummy M so that H is major component
A IROW B	any	B vector replaces Ith row of A
MANDH I	any	displays M and H for explicit I-vector entries
Z ← P MFNH H		returns P vector as FN's of given H vector
Z ← MFNHA J	C	returns TM as FN's of each HA[J;]
Z ← MFNI I		returns TGM[I;] as FN's of corresponding H[I;]
NAMENT	C	displays current HA roots, classes, TNS, and FRD values
A PRTNMS B	C	displays subset of NAMENT result for A ∊ B where A = FRD, TNS, etc. and B is a vector of desired values
Z ← SETNAMES H		constructs Root/Class list according to structures in given H array
SKIP	any	lets current H[I;] be undefined (no H for this TM)

Fig. 14-3. Reference sheet for the information and utility functions available in the harmonization/melodization workspace.

```
     ∇ M←I DVLP KER;S;M1;J;K
[1]    S←TONSYS SCLIT KER
[2]    J←⌈(K←|+/I)÷2
[3]    M1←S MTF I
[4]    M←M1,M1,M1,M1 CNTUS I
[5]    M←M CNTUS(−K),I
[6]    M←M CNTUS(−J),(J⍴1),(J⍴¯1),J⍴1
[7]    S←SCL⍳¯1↑M
[8]    M1←S MTF−I
[9]    M←M,M1,M1 CNTUS(−J),((J+K)⍴1),−J
[10]   S←SCL⍳¯1↑M
[11]   M1←S MTF 0 ¯1 1 ,J,(−K),K−J
[12]   M←M,M1 CNTUS(K⍴0),¯1,K,(−K),1
[13]   S←SCL⍳¯1↑M
[14]   M1←S MTF((K−J)⍴1),J⍴¯1
[15]   M←M,M1,M1,(S MTF(K⍴0),1+K),M1
[16]   M←M CNTUS(2+K),(1+K)⍴¯1
     ∇

     ∇ Z←T SCLIT I;J
[1]    J←I BRK T
[2]    Z←SCLI J
     ∇

     ∇ Z←S MTF I
[1]    Z←SCL[+\S,I]
     ∇

     ∇ Z←M CNTUS I;K
[1]    K←SCL⍳¯1↑M
[2]    →(K<⍴SCL)↑L1
[3]    K←SCL⍳(¯1↑M) FITSCL SCL
[4]   L1:Z←(¯1↓M),K MTF I
     ∇
```

Fig. 14-4. DVLP and auxiliary functions used to compose the thematic material for the results shown in Fig. 13-3 and 13-4.

"attack" vector, ATK, defined here as

ATK ←2 4 4 4 3 · · ·

directs the way the 81 notes of the melody are to be assigned to 24 successive segments for harmonization.

BEGINNING A SESSION

The complete listing of the first session to be

discussed is shown at the end of the chapter in Listing 14-1. I will not go into such great and agonizing detail here as in previous sessions. Questions concerning the various operations that are not discussed can be answered by referring to the program descriptions in the appendix—providing those questions ask what rather than why. The latter type of question can be answered superficially by saying that this example was meant to show typical operations and, simultaneously, to display the way "common" harmonic continuity can be achieved by selecting chords from the lower-order categories defined in the previous chapter. I will describe the features of this session that did not appear in the listings of the earlier harmonization sessions.

HOFM is now called without giving explicit arguments. The melodic and harmonic-attack vectors to be used must have the global names, M and ATK, respectively. Typing "**HOFM**" causes the program to display the current settings of some other global variables which may be changed by the user at any time during or between sessions. The first of these to appear,

```
NPH: 3
```

indicates that, unless a change such as **NPG ← 5** is explicitly directed, tensions will be computed by referring each target melody to only the first three Notes Per Harmonic structure in the HA (harmonic alternative) array.

Following that, we see

```
GSIG CURRENTLY SET TO
17 7 15 · · ·
```

stating that the variable GSIG contains the values shown. These are row indices of the structure array, the list of 36 seven-note harmonic structures. That array now has the global name, SIGMA; Schillinger called such complete tonalities *sigma structures.* Until GSIG is changed, the program will behave as though the structure array was limited to just these rows and in this order. That is, "the" structure array in use will always be **SIGMA [GSIG;]**.

The next message

```
RDUP AND NDUP SET TO 1 AND 3
```

says that, as these variables are currently set, *R*oot-tone *DUP*lication may not occur within any single (1) change of tonality, i.e., two adjacent chords cannot have the same root. Similarly, *N*o *DUP*lication of the same chord (root and structure) will be allowed within three successive changes of tonality.

After these messages, the program appears to function pretty much as did the earlier version, except for an obvious cosmetological improvement. At the beginning of each I-time, a line of special symbols is printed, making it easier for the user to keep track of his work and for us to refer to particular sections in the session. And, speaking of special symbols, even the sharp sign is now available (although it appears somewhat strange under photo-reduction).

OTHER NEW FEATURES

SKIP. For the very first chord, I = O, function **SKIP** was called. This allows the chord for the cur-

```
        M←¯3+1 1 1 DVLP 1 2 2 2 2

        M
¯3 ¯2  0  2 ¯3 ¯2  0  2 ¯3 ¯2  0  2 ¯3 ¯2  0  2  4  6  7  2
 4  6  7 '4  6  7  6  4  6  7  7  6  4  2  7  6  4  2 ¯2  0
 2  4  6  7  4  4  4  2  4  7  2  4  4  4  4  2  7  2  4  4
 6  4  2  4  6  4  2  4  4  4  4 11  4  6  4  2 11  9  7  6
 4

         ATK←2 4 4 4 3 4 3 5 1 4 4 4 3 6 1 6 1 4 4 4 1 4 4 1
```

Fig. 14-5. Construction of the M and ATK vectors for the terminal sessions in Listing 14-1, 14-2, and 14-3.

rent I value to remain undefined. In this particular case I had no intention of returning later to fill in a suitable chord. The basic motif for this melody contains four consecutive scalewise steps, and I decided to treat that pattern so that the first two notes come before the strong beat of the meter (as pick-up notes) without accompaniment.

STRUC and associated globals. At S-time for the next two harmonizations (I = 1 and I = 2), the function named **STRUC** was used. This function keeps only those chords in the array of harmonic alternatives whose structure indices (row numbers in SIGMA) are in the right argument. By establishing global variables such as MA, MI, and MI7 to be used for that argument, the user needn't remember how to associate particular chords with row indices, i.e., MA can contain the proper index for "the" major structure, MI for the minor, and MI7 for the minor seventh. In the appendix you'll find MA has the value 17, and then, by counting down to that row in SIGMA (beginning the count with zero), you'll see the structure being singled out has the intervals 4 3 4 3 4 3.

FRDGRP and freedom (FRD). By skipping the first (zeroth) chord, the first tonality actually assigned was for I = 1. Not until I = 2, then, could our hypothesis on harmonic continuity first be applied. At C-time for that case, function **FRDGRP** was invoked to display how the harmonic alternatives populated the six categories. Recall, those categories indicate the number of notes that are free to change on going from one chord to the next. For this reason, the category indices are called *freedom values*, i.e., a harmonic alternative that lies in category number one is said to have its freedom (FRD) equal to one. Further, since this quantity relates pairs of chords, it is possible for each harmonic alternative to carry two freedom values. That is, in a case where a target melody is purposely left unharmonized until its preceding and following chords have been established, a negative freedom value will link each harmonic alternative to the previous chord and a positive value of freedom will classify its relation to the next chord. Thus, if a possible harmony has freedom values of ¯2 and 1, it contains just two tones not in the previous seven note structure and one tone not in the next.

Here then, after calling **FRDGRP**, the FReeDom GRouPings or populations in the categories are shown to be

```
  0 1 2 3 4 5
- 1 2 0 0 0 0
```

This tells us that only three harmonic alternatives have met the specifications to this point (they must have MI or MI7 structures), and one of them is in the zeroth category (FRD = 0) while the other two have freedoms of one. Note the minus sign preceding the values in second row here; it specifies that the population distribution in this row relates the HA structures to the preceding chord (I = 1). Because I = 3 has not yet been defined, there is no distribution shown preceded by a plus sign.

KEPS. Immediately following the display of freedom groupings, the statement

```
FRD KEPS 0
```

used the **KEPS** function to keep those harmonic alternatives whose FRD values were members of the right argument (zero here). **KEPS** is a "bilingual" mnemonic: K for Keep and EPS for the epsilon operator (ε) which means "is a member of." The conformability requirements for the left and right arguments, as well as some suggestions for what those arguments might be, are described in the appendix.

PRTNMS. For the next chord (I = 3), I again wanted only a single note to change, so instead of asking to see the entire distribution, I requested just a list of those with FRD equal to one:

```
FRD PRTNMS 1
```

The *PR*in*T* *N*a*M*e*S* function then showed four structures, listing the root tone, class, tension (TNS), and freedom (FRD) for each. To make sure one of these would be selected, I entered

```
FRD KEPS 1
```

Note that the C-time condition will by default choose the one with minimum tension.

NAMENT. The harmonic alternatives for I = 4 on entering C-time were found to consist of 15 structures distributed in the first three freedom categories. Using

```
STRUC A7
```

to remove all but the augmented-seventh structures reduced HA to just five structures. **FRDGRP** showed their distribution, and **NAMENT** listed them in the same format as described above for **PRTNMS**. **PRTNMS** displays a selected subset of HA, while **NAMENT** (with no arguments) shows the entire collection of harmonic alternatives (root, class, tension, and freedom).

EQH and FITMALL. Two new features were used in establishing the next setting (I = 5). First, at S-time

```
0 EQH 1
```

commands this chord to be the same as (EQual Harmony) that for I = 1 with no (0) transposition. Then, at C-time

```
FITMALL FN 1 3 5 7 9 11 13
```

causes the notes in the target melody to be adjusted as needed to fit the entries in the harmonic structure given by the right argument of **FITMALL**. The function FN allows the user to think in terms of musical indices (the odd numbers from one to 13) instead of column indices (one to seven) in the harmonic structure. When the refitting was complete, the changed target melody was shown in the message:

```
TM  RESET  TO  5 7 2 4.
```

The Quad List. While not a completely new feature, the display produced through use of the quad symbol at S-time for I = 7 is of interest. Typing just a quad symbol causes all the chords selected so far to be enumerated by root, class, and the actual numeric values in the array, H. Note in particular, the first chord (remember, I = 0 was **SKIP**ped) is represented in the H array by a 36 followed by seven zeroes. That 36 points to a nonexistent row of SIGMA, whose row indices stop at 35.

CLASS and related variables. Just as **STRUC** can be used to limit harmonic choices to particular structures, a function named **CLASS** operates in a parallel way, restricting selections to particular classes. The global variable CLSNDX is a vector that shows which of the twelve classes applies to each of the 36 structures. Other globals, such as MAC and MIC, can be preset to let the user refer to MAjor Classes or MInor Classes so that he doesn't need to memorize or count to find the corresponding class indices. Thus, for I = 10 at S-time,

```
CLASS MAC,MIC
```

was used to allow only structures 6, 8, 10, 3, and 2.

PRTL and BACK. For the twelfth harmonization (I = 11), after finding that two structures have FRD = 1 and both have very low tensions, the statement

```
PRTL 1 3 5 7
```

was entered. This causes tension values of the target melody with respect to the harmonic alternatives to be recomputed. Instead of calculating the tension using the first NPH notes in the structures (**HA[;1 + ⍳NPH]**), the harmonic functions specified in the right argument will be used (**HA[;FN 1 3 5 7]**). I call this a "partial" tension, hence the name **PRTL**. (In the appendix, the listing of this function shows this description is correct, but the description of the function in the appendix is in error. The function was updated so that FN would not be needed explicitly in the right argument, but I did not update the explanation to reflect this change! I must decide whether to keep this change and update all the other functions, or remove it so that column indices of HA are used in a consistent way throughout the system.) Upon seeing how little dif-

ference this made on the tensions of the two structures, I typed

BACK 1.

Immediately, the target melody and the regular S-time message were reprinted. This function takes the procedure back to the beginning of the current time if the right argument is zero, to the beginning of the previous time if that argument equals one, or to the start of I-time for a larger argument. The function was called at C-time here, which explains why the argument, one, reinitialized S-time.

I believe the interested reader, who studies the details of this session now, will begin to feel competent about using the system.

A MORE REALISTIC TREATMENT

A somewhat more authentic, less didactic approach was taken in the next session (Listing 14-2). First, the same target melody was handled with more musical feeling. Again, there is more detail here than can be expounded without boring both of us to lachrymosity, but with the aid of the appendix and what has already been said, careful study of Listing 14-2 should prove more valuable than my explanation and further increase the reader's competence.

Next a contrasting B theme was constructed to provide a more musical result and also to demonstrate a new approach to melodization. Here, despite the availability of the program descriptions, I will assume that techniques unlike any previously described deserve a bit of elaboration.

PREPARING FOR MELODIZATION

Before beginning this part of the session, I made sure the melody and harmony constructed in the previous session (Listing 14-2) were well preserved. Notice that in response to the "**OK?**" message I put them away for safekeeping under the names MM and HH. Now, instead of concocting another painstaking melodic development, a program named **HPATTERN** was called.

In the session dialogue (Listing 14-3), you can see that function **HPATTERN** responds by asking the user to enter a vector consisting of pairs of numbers in which the first member of each pair is to specify a number of harmonic structures (**H'S**) and the second must state the number of beats or time units each chord will occupy (**T-UNITS/H**). I responded by entering three pairs of values:

6 4 4 2 8 4

meaning that I wanted the first six chords each to have durations of four "beats," the next four structures were each to occupy two beats, and the last eight chords were each to be four beats in length.

The program then told me to enter rhythmic information for a (nonexistent) melody to go with the first six harmonizations:

ENTER ATK PATTERN FOR 6 H'S

My four-entry response

4 3 4 1,

indicated that I expected those six chords to harmonize 4, 3, 4, 1, 4, and 3 melody notes, respectively. (Notice the recycling here to acquire six values from the four that were entered.)

Without pausing to find out what those melodic tones might be, the program requested similar information for the second group of chords, the four chords that are each to have two beat durations. My reply,

2 1

was followed by one more such request, this one for the last group of eight structures. Following my response, the program declared that something akin to the M and ATK variables needed by the **HOFM** algorithm had been created—"dummy" variables M0 and ATK0! Then, as a last reminder, a tabular listing of the number of chords, H, and the number of beats for each chord, B, was shown.

"SIMULTANEOUS" MELODIZATION AND HARMONIZATION

I accepted the dummy variables as the real thing by stating

M ← M0
ATK ← ATK0

Then **HOFM** was called, and it gave forth its usual stream of information before opening I-time for chord number zero. You might notice here that GSIG contains all 36 structures, but ordered by personal preference. This same setting was used in the previous session. The program treats structures in their given order. If, at the end of C-time, a tie occurs for "best" harmonization, the first chord in the list with all the "best" properties will be chosen. So the imposition of preference here gives the user some small voice as a tie-breaker.

In I-time, I asked to see the ATK vector which had been pieced together with the aid of the **HPATTERN** function. Then, to be sure I didn't miscount the number of entries in this vector, I asked to see the shape of ATK. Now, observing that there are 18 harmonizations to be performed, I established for my own records a "**FORM**" and a "**PLAN**." The **FORM** tells me to keep in mind that the melody for this session is to conform to the "aabb" pattern with each section containing four chords except the second "a" section, which will hold four plus the extra pair one finds when dividing 18 by four:

FORM ← 'A(4) A(6) B(4) B(4)'

Similarly, **PLAN** ties memory strings on chord numbers 0, 4, 10, and 14 as starting points for the four sections of **FORM**:

PLAN ← 0 4 10 14

On entering S-time, we get our first indication of the unimaginative outline for the dummy melody. For four attacks, **HPATTERN** selected the first four integers, zero through three. Function **UPTM** was then used to *UP*date this Taget Melody to something more musical and more related to the MM melody:

UPTM (¯1↑⍙,MM) NVRT 4↑⍙,MM

This will invert (**NVRT**) the first four notes of MM (4↑⍙,MM) and transpose that pattern to begin on the last pitch of MM (¯1↑⍙DN,MM). I asked for the updated target melody to be displayed (⎕ TM) and was shown

3 2 0 ¯2

The next three commands

XF ← 1
XP ← TM
S ← MA

specified that the root of the harmony should not be in the target melody, and the chord should be a major structure. Fortunately, one chord was found to satisfy these requirements and, since harmonization only requires a single satisfactory chord, on to I = 1.

The target melody chosen for three attacks was seen to be just as mechanical as for four:

TM = 0 1 2

But this time I let the harmony set the pace by ordering:

¯4 EQH 0

i.e., the same harmonic structure as for I = 0 is to be used here, but transposed down four semitones with no regard for what it is supposed to harmonize. Then at C-time, function **MLDZ** was called to *MeLoDiZe* this chord with the proper number of attacks. The left argument, 0 ¯1, indicates that a reference point for the melodization should be chosen by transposing the last pitch of the previous target melody (TM for I = 0) through the interval zero or minus one, which ever places it in this structure. If neither interval gives us a legitimate pitch, the closest note in the structure to that last pitch will be used. With the structure compressed into a scale, the right argument, 1 3 1, describes how this melodic segment is to proceed from the reference tone in *scalewise* intervals. And then the message

TM RESET TO ¯1 4 6

displays the resulting melodization for this chord.

For the next harmonization (I = 2), at S-time I looked over the output from the previous session and decided that the four-note motif used for I = 17 in that session should serve as a stylistic bond here. Always aware of the difference in our memory circuits, I asked to see what the computer thought **MM[17;]** contained. Its report **(3 5 3 1)** agreed with my own finding, so **UPTM** was called to force this to become the current target melody. At C-time, the 22 structures with a freedom value of 2 were singled out by the statement

FRD KEPS 2.

Then, I asked only those with structure type **MI7** to step forward. One fully qualified member was present in the ranks, and so on to I = 3.

The single melodic target pitch for this section, chosen as zero by **HPATTERN**, was changed to the first pitch of the previous motif, **TGM[I-1;0]**. The structure specification, **S←MA** limits harmonic alternatives to row MA of the SIGMA array. Four possibilities were found, all with relatively low tension, and the one with the highest **(CND←⌈/TNS)** was chosen.

With apologies to those who take delight in such endless descriptions, and with mercy for the greater number who do not, I will end this one here. The complete listing of the session will now prove more informative if the reader continues the analysis on his own, referring whenever necessary to the program listings and descriptions in the appendix. The enormous amount of freedom in this limited logical setting will become evident when you have enough familiarity with the available functions and variables to guide your own sessions, make your own decisions, and change or add functions according to your own needs.

Listing 14-1. Listing of a session in which the lower freedom categories were used.

```
HOFM
NPH: 3
GSIG CURRENTLY SET TO
17 7 15 5 23 30

RDUP AND NDUP SET TO 1 AND 3

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=0 ?
SKIP

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=1 ?

TM=0 2 ¯3 ¯2
S, P, XP, F OR XF (OR □) ?
STRUC MA

CND=⌊/TNS?  (CND,TNS,HA,H?)

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=2 ?

TM=0 2 ¯3 ¯2
S, P, XP, F OR XF (OR □) ?
STRUC MI,MI7

CND=L/TNS?  (CND,TNS,HA,H?)
```

```
FRDGRP
   0 1 2 3 4 5
-  1 2 0 0 0 0
FRD KEPS 0

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=3 ?

TM=0 2 ¯3 ¯2
S, P, XP, F OR XF (OR ⎕) ?

CND=L/TNS?  (CND,TNS,HA,H?)
FRD PRTNMS 1
 C     LG7      22    1
 C     MI7     121    1
 F     7+3     110    1
 D#    MA7     220    1
FRD KEPS 1

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=4 ?

TM=0 2 4
S, P, XP, F OR XF (OR ⎕) ?

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
    0  1  2  3  4  5
-   1  3 11  0  0  0
STRUC A7
FRDGRP
   0 1 2 3 4 5
-  0 1 4 0 0 0
NAMENT
 F#    +7      22  2
 E     +7      11  2
 D     +7      22  1
 A#    +7      22  2
 G#    +7      11  2

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=5 ?
TM=6 7 2 4
S, P, XP, F OR XF (OR ⎕) ?
0 EQH 1

CND=L/TNS?  (CND,TNS,HA,H?)
FITMALL FN 1 3 5 7 9 11 13
TM RESET TO 5 7 2 4

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=6 ?

TM=6 7 4
```

```
S, P, XP, F OR XF (OR D) ?
0 EQH 2

CND=L/TNS?  (CND,TNS,HA,H?)
FITMALL FN 1 3 5 7 9 11 13
TM RESET TO 5 7 4

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=7 ?

TM=6 7 6 4 6
S, P, XP, F OR XF (OR D) ?
D
SKIP          36  0  0  0  0  0  0  0
 A#   MA7     17 10  2  5  9  0  4  7
 G    MI7      5  7 10  2  5  9  0  4
 C    LG7     15  0  4  7 10  2  6  9
 E    +7      23  4  8  0  2  6 10 11
 A#   MA7     17 10  2  5  9  0  4  7
 G    MI7      5  7 10  2  5  9  0  4
STRUC L7

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 0 0 1 0
FRD KEPS 1

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=8 ?

TM=7
S, P, XP, F OR XF (OR D) ?
STRUC L7

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 1 4 0 1
NAMENT
 F    LG7      11   3
 D#   LG7       0   3
 C#   LG7     110   5
 A#   LG7      10   2
 A    LG7       1   3
 G    LG7       0   3


∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=9 ?

TM=7 6 4 2
S, P, XP, F OR XF (OR D) ?
STRUC MA,MI
```

```
CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 0 1 2 0
NAMENT
 G    MA7     110    4
 G    MI      110    3
 C    MA7     121    4

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=10 ?

TM=7 6 4 2
S, P, XP, F OR XF (OR D) ?
CLASS MAC,MIC

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  1 2 0 0 0 0
NAMENT
 C    MA7     121    1
 A    MI7      22    1
 E    MI7     111    0

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=11 ?

TM=¯2 0 2 4
S, P, XP, F OR XF (OR D) ?

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 2 6 5 0 0
FRD PRTNMS 1
 C     LG7      12    1
 G     MI       21    1
PRTL 1 3 5 7
FRD PRTNMS 1
 C     LG7      11    1
 G     MI       22    1
BACK 1

TM=¯2 0 2 4
S, P, XP, F OR XF (OR D) ?
CLASS MIC,MAC

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 2 1 0 0
NAMENT
 A#    MA7     121    2
```

```
 G     MI       21   1
 G     MI7      21   2
 F     MI      121   3
FRD KEPS 1

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=12 ?

TM=6 7 4
S, P, XP, F OR XF (OR □) ?

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  1 3 2 2 0 0
NAMENT
 C   MA7     110   1
 C   LG7     110   0
 C   +7      111   1
 A   LG7      11   3
 A   7+3      11   2
 E   MI      110   3
 E   MI7     110   2
 D   7+3     110   1
FRD KEPS 0

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=13 ?

TM=4 4 2 4 7 2
S, P, XP, F OR XF (OR □) ?
STRUC MA

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 2 0 0 0
NAMENT
 F     MA7     121   2
 A#    MA7     120   1
 G     MA7      10   2
PRTL 1 3 5 7
NAMENT
 F     MA7      22   2
 A#    MA7     121   1
 G     MA7      11   2
PRTL 1 3 5 7 9
NAMENT
 F     MA7      11   2
 A#    MA7     121   1
 G     MA7      11   2

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=14 ?
```

```
TM=4
S, P, XP, F OR XF (OR D) ?
CLASS 5

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 2 1 1 1
NAMENT
 A#   LG7    110   2
 A    LG7      0   3
 G    LG7     10   1
 F#   LG7      1   5
 E    LG7      0   4
 D    LG7     11   2
FRD KEPS 1

*O*O*O*O*O*O*O*O*O*O*O*O*O*O*O*
I=15 ?

TM=4 4 4 2 7 2
S, P, XP, F OR XF (OR D) ?
STRUC MI7

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 2 1 0 0 0
NAMENT
 A   MI7     12   2
 E   MI7      1   1
 D   MI7    121   1

*O*O*O*O*O*O*O*O*O*O*O*O*O*O*O*
I=16 ?

TM=4
S, P, XP, F OR XF (OR D) ?
STRUC L7

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 2 0 2 0
NAMENT
 A#   LG7    110   4
 A    LG7      0   1
 F#   LG7      1   4
 D    LG7     11   2
 C    LG7      0   2

*O*O*O*O*O*O*O*O*O*O*O*O*O*O*O*
I=17 ?

TM=4 6 4 2
```

```
S, P, XP, F OR XF (OR D) ?
STRUC MA

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 2 0 0 0
NAMENT
 D    MA7      11    2
 C    MA7     121    2
 G    MA7     110    1

oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOo
I=18 ?

TM=4 6 4 2
S, P, XP, F OR XF (OR D) ?
CLASS MIC

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  1 2 1 1 0 0
NAMENT
 G    MI      110    3
 B    MI       11    1
 B    MI7      11    0
 A    MI       21    1
 A    MI7      21    2
FRD KEPS 0

oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOo
I=19 ?

TM=4 4 4 4
S, P, XP, F OR XF (OR D) ?
STRUC L7

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 2 1 2 1 0

NAMENT
 E     LG7       0    1
 D     LG7      11    1
 C     LG7       0    3
 A#    LG7     110    4
 G     LG7      10    2
 F#    LG7       1    3
FRD KEPS 1

oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOo
I=20 ?
```

```
□¯1↑⍴ATK
23
GSIG←UNIQ GSIG,(CLSNDX∊L7C)/⍳36

TM=11
S, P, XP, F OR XF (OR D) ?
CLASS L7C

CND=⌊/TNS?  (CND,TNS,HA,H?)
FRDGRP
    0  1  2  3  4  5
-   0  4 13 20 14  3
FRD KEPS 2
CND←⌈/TNS
NAMENT
 A#    LG7     1000     2
 A     7+3       10     2
 G#    +7       100     2
 G     LG7        0     2
 G     LG7        0     2
 F#    7+3        0     2
 F#    7+3        0     2
 D     LG7       10     2
 C#    LG7        1     2
 C#    LG7        1     2
 C#    7+3        1     2
 B     7+3        0     2
 B     7+3        0     2

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=21 ?


TM=4 6 4 2
S, P, XP, F OR XF (OR D) ?
STRUC MI7

CND=⌊/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 0 2 0 0
NAMENT
 A   MI7     21   3
 E   MI7    111   3

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=22 ?

TM=11 9 7 6
S, P, XP, F OR XF (OR D) ?
DL7C
5 7 9
CLASS 9
```

```
CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  1 2 0 0 0 0
FRD KEPS 1


*O*O*O*O*O*O*O*O*O*O*O*O*O*O*
I=23 ?

TM=4
S, P, XP, F OR XF (OR D) ?
STRUC MA

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 2 2 1 0
NAMENT
 G     MA7     10   2
 F     MA7    100   2
 E     MA7      0   4
 C     MA7      0   1
 A#    MA7    110   3
 A     MA7      0   3
FRD KEPS 2
OK?
TO KEEP H, ENTER: 'NAME←H', OTHERWISE CR

-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-

ORIGINAL TARGET M:
   ¯3 ¯2
   0 2 ¯3 ¯2
   0 2 ¯3 ¯2
   0 2 ¯3 ¯2
   0 2 4
   6 7 2 4
   6 7 4
   6 7 6 4 6
   7
   7 6 4 2
   7 6 4 2
   ¯2 0 2 4
   6 7 4
   4 4 2 4 7 2
   4
   4 4 4 2 7 2
   4
   4 6 4 2
   4 6 4 2
   4 4 4 4
   11
   4 6 4 2
```

```
     11  9  7  6
     4

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
M:   ¯1:   A      A#
H:

M:   0:   C      D   ¯1A      A#
H:        A#     MA7

M:   0:   C      D   ¯1A      A#
H:        G      MI7

M:   0:   C      D   ¯1A      A#
H:        C      LG7

M:   0:   C    D    E
H:        E     +7

M:   0:   F    G    D    E
H:        A#    MA7

M:   0:   F    G    E
H:        G     MI7

M:   0:   F#    G     F#    E     F#
H:        C     LG7

M:   0:   G
H:        D#    LG7

M:   0:   G     F#    E     D
H:        G     MA7

M:   0:   G     F#    E     D
H:        A     MI7

M:   ¯1:   A# +1C     D     E
H:        G     MI

M:   0:   F#    G     E
H:        C     LG7

M:   0:   E    E    D    E    G    D
H:        F     MA7

M:   0:   E
H:        G     LG7

M:   0:   E    E    E    D    G    D
H:        E     MI7

M:   0:   E
H:        A     LG7
```

```
M:   0:  E     F#    E     D
H:       D     MA7

M:   0:  E     F#    E     D
H:       B     MI7

M:   0:  E    E    E    E
H:       E     LG7

M:   0:  B
H:       A#    LG7

M:   0:  E     F#    E     D
H:       A     MI7

M:   0:  B     A     G     F#
H:       D     7+3

M:   0:  E
H:       G     MA7
```

Listing 14-2. The session treating the same M and ATK vectors but mixing the ideas of the hypothesis on order with additional musical considerations.

```
HOFM
NPH: 3
GSIG CURRENTLY SET TO
 17 15  7  5  0 25 30 32 20 16  9  6
  3  1  8 12 18 26 27 34  2 11 33 10
 21 28 35 13  4 23 14 29 24 31 19 22

RDUP AND NDUP SET TO 1 AND 3

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=0 ?
SKIP

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=1 ?

TM=0 2 ¯3 ¯2
S, P, XP, F OR XF (OR ⎕) ?
XP←TM
XF←1

CND=L/TNS?  (CND,TNS,HA,H?)
⎕⍴HA
14 8
CLASS MIC
⎕⍴HA
4 8
```

```
NAMENT
 G     MI      121    6
 D♯    MI      210    6
 G     MI      121    6
 G     MI7     121    6

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=2 ?

TM=0 2 ¯3 ¯2
S, P, XP, F OR XF (OR □) ?
STRUC MA

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 1 1 0 0 0
NAMENT
 A♯    MA7     111    1
 D♯    MA7     220    2
3 F2FN 1

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=3 ?

TM=0 2 ¯3 ¯2
S, P, XP, F OR XF (OR □) ?
STRUC A7

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 0 1 0 0

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=4 ?

TM=0 2 4
S, P, XP, F OR XF (OR □) ?
CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
    0  1  2  3  4  5
-   1 18 26 10  0  0
FRD KEPS 0

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=5 ?
□
SKIP          36  0  0  0  0  0  0  0
 G     MI      7  7 10  2  6  9  0  4
 D♯    MA7    17  3  7 10  2  5  9  0
 D     +7     23  2  6 10  0  4  8  9
 A♯    +MA7   25 10  2  6  9  0  4  8
```

```
TM=6 7 2 4
S, P, XP, F OR XF (OR D) ?

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
     0  1  2  3  4  5
-    0  5 16  9  0  0
FRD KEPS 1
NAMENT
 C    LG7    121   1
 C    +7     122   1
 G#   +MA7   122   1
 G    MI     110   1
 E    -MI7   111   1

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=6 ?


TM=6 7 4
S, P, XP, F OR XF (OR D) ?
0 EQH 2

CND=L/TNS?  (CND,TNS,HA,H?)
DHA HTNSNM TM
 2100    0 1210
NAMENT
 D#   MA7    1100    2
FITM FN 7 9 11
TM RESET TO 5 7 5

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=7 ?

TM=6 7 6 4 6
S, P, XP, F OR XF (OR D) ?
STRUC L7

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 1 0 1 0
FRD KEPS 2

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=8 ?

TM=7
S, P, XP, F OR XF (OR D) ?
DL7C
5 7 9
CLASS 5

CND=L/TNS?  (CND,TNS,HA,H?)
```

```
FRDGRP
   0 1 2 3 4 5
-  0 0 6 8 3 1
⎕TNS
1000 11 100 0 0 0 110 110 110 10 10 10 1 1 1 0 0 0
CND←⌈/TNS

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=9 ?

TM=7 6 4 2
S, P, XP, F OR XF (OR ⎕) ?
¯1 OVRIDS MA

CND=⌊/TNS?  (CND,TNS,HA,H?)
⎕HA HTNSNM TM
    0 2110    0    0
FITMALL FN 1 3 5 7 9 11 13
TM RESET TO 7 5 4 2

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=10 ?

TM=7 6 4 2
S, P, XP, F OR XF (OR ⎕) ?
3 OVRIDS MA
TM←⍎TGM[I-1;]

CND=⌊/TNS?  (CND,TNS,HA,H?)
⎕TNS
1211

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=11 ?

TM=¯2 0 2 4
S, P, XP, F OR XF (OR ⎕) ?
⎕MAC
6 8 10
4 OVRIDC 6

CND=⌊/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 0 3 2 0
FRD KEPS 4
NAMENT
 C    MA7    12  4
 C    MA7    12  4
FITM FN 1 3 5 7 9
TM RESET TO ¯1 0 2 4

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=12 ?
```

```
⎕TGM[I+⍳5;]
6 7 4
4 4 2 4 7 2
4
4 4 4 2 7 2
4
I←14

TM=4
S, P, XP, F OR XF (OR ⎕) ?
STRUC L7S9
F←FN 9
CND=L/TNS?  (CND,TNS,HA,H?)
∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=12 ?
I←13

TM=4 4 2 4 7 2
S, P, XP, F OR XF (OR ⎕) ?
⎕L7C
5 7 9
CLASS 9

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
+  0 0 3 4 2 0
FRD KEPS 2

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=12 ?

TM=6 7 4
S, P, XP, F OR XF (OR ⎕) ?
¯2 OVRIDC 6

CND=L/TNS?  (CND,TNS,HA,H?)
⎕TNS
1120 1120 1120 1120 1120
FITMALL FN 1 3 5 7 9 11 13
TM RESET TO 5 7 4
FRDGRP
   0 1 2 3 4 5
-  0 0 1 3 1 0
+  0 0 2 3 0 0
 NET ∆ = 3
NAMENT
 A♯   MA7    1120    2    3
 A♯   MA7    1120    3    3
 A♯   MA7    1120    3    3
 A♯   MA7    1120    3    2
 A♯   MA7    1120    4    2
NARROW 4
```

```
∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=15 ?

TM=4 4 4 2 7 2
S, P, XP, F OR XF (OR ⎕) ?
⎕TGM[I+⍳5;]
4 4 4 2 7 2
4
4 6 4 2
4 6 4 2
4 4 4 4
⎕TGM[I-2 1;]
4 4 2 4 7 2
4
¯1 EQH I-2
(I↓⍳⍴ATK) CHNGTM ¯1
TM RESET TO 3 3 3 1 6 1

CND=⌊/TNS?  (CND,TNS,HA,H?)
⎕HA HTNSNM TM
 0 0 0 0 0 0

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=16 ?
TM=3
S, P, XP, F OR XF (OR ⎕) ?
¯1 EQH I-2

CND=⌊/TNS?  (CND,TNS,HA,H?)
HA HTNSNM TM

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=17 ?

TM=3 5 3 1
S, P, XP, F OR XF (OR ⎕) ?
STRUC MA

CND=⌊/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 0 2 1 0
FRD KEPS 4

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=18 ?


TM=3 5 3 1
S, P, XP, F OR XF (OR ⎕) ?
2 EQH I-1

CND=⌊/TNS?  (CND,TNS,HA,H?)
⎕HA HTNSNM TM
```

```
 0 0 0 0

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=19 ?

TM=3 3 3 3
S, P, XP, F OR XF (OR ⎕) ?
0 EQH I-2

CND=L/TNS?  (CND,TNS,HA,H?)
⎕HA HTNSNM TM
 0 0 0 0

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=20 ?

TM=10
S, P, XP, F OR XF (OR ⎕) ?
6 EQH I-3

CND=L/TNS?  (CND,TNS,HA,H?)
⎕HA HTNSNM TM
1101
FITM FN 1 3 5 7 9 11 13
TM RESET TO 9

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=21 ?
⎕⍴ATK
24
I←23

TM=3
S, P, XP, F OR XF (OR ⎕) ?
0 EQH 17

CND=L/TNS?  (CND,TNS,HA,H?)
⎕HA HTNSNM TM
0

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=21 ?
I←22

TM=10 8 6 5
S, P, XP, F OR XF (OR ⎕) ?
0 EQH 18

CND=L/TNS?  (CND,TNS,HA,H?)
⎕HA HTNSNM TM
    0     0 2101     0
FITMALL FN 1 3 5 7 9 11 13
TM RESET TO 10 8 7 5
```

```
∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=21 ?

TM=3 5 3 1
S, P, XP, F OR XF (OR D) ?

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
     0  1  2  3  4  5
-    0  0  4 16 22  7
+    3 10 23 12  1  0
 NET ∆ = 4
DL7C
5 7 9
CLASS 5
FRDGRP
   0 1 2 3 4 5
-  0 0 0 1 1 0
+  0 1 0 1 0 0
 NET ∆ = 4
NAMENT
 B     LG7     121    4    3
 D⍕    LG7      12    3    1
OK?
TO KEEP H, ENTER: 'NAME←H', OTHERWISE CR
MM←TGM
HH←H
_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_‾_

ORIGINAL TARGET M:
   ¯3 ¯2
   0 2 ¯3 ¯2
   0 2 ¯3 ¯2
   0 2 ¯3 ¯2
   0 2 4
   6 7 2 4
   6 7 4
   6 7 6 4 6
   7
   7 6 4 2
   7 6 4 2
   ¯2 0 2 4
   6 7 4
   4 4 2 4 7 2
   4
   4 4 4 2 7 2
   4
   4 6 4 2
   4 6 4 2
   4 4 4 4
   11
   4 6 4 2
   11 9 7 6
   4
```

```
∘0∘0∘0∘0∘0∘0∘0∘0∘0∘0∘0∘0∘0∘0∘0∘

M:  ¯1:  A    A#
H:

M:  0:  C    D  ¯1A    A#
H:      G    MI

M:  0:  C    D  ¯1A    A#
H:      D#   MA7

M:  0:  C    D  ¯1A    A#
H:      D    +7

M:  0:  C   D   E
H:      A#   +MA7

M:  0:  F#   G    D    E
H:      G    MI

M:  0:  F   G   F
H:      D#   MA7

M:  0:  F#   G    F#   E    F#
H:      C    LG7

M:  0:  G
H:      F#   LG7

M:  0:  G   F   E   D
H:      F    MA7

M:  0:  G   F   E   D
H:      G#   MA7

M:  ¯1:  B +1C   D   E
H:      C    MA7

M:  0:  F   G   E
H:      A#   MA7

M:  0:  E   E   D   E   G   D
H:      E    7+3

M:  0:  E
H:      C#   LG7

M:  0:  D#   D#   D#   C#   F#   C#
H:      D#   7+3

M:  0:  D#
H:      C    LG7

M:  0:  D#   F    D#   C#
```

```
H:        B     MA7

M:   0:   D#    F     D#    C#
H:        C#    MA7

M:   0:   D#    D#    D#    D#
H:        B     MA7

M:   0:   A
H:        F     MA7

M:   0:   D#    F     D#    C#
H:        D#    LG7

M:   0:   A#    G#    G     F
H:        C#    MA7

M:   0:   D#
H:        B     MA7
```

Listing 14-3. Listing of a session in which melody and harmony for a "B" theme are composed simultaneously.

```
HPATTERN

ENTER VECTOR OF PAIRS OF (NO. OF H'S, NO. OF T-UNITS/H)
6 4 4 2 8 4
ENTER ATK PATTERN FOR 6 H'S
4 3 4 1
ENTER ATK PATTERN FOR 4 H'S
2 1
ENTER ATK PATTERN FOR 8 H'S
4 4 3 1
DUMMY ATK0 AND M0 NOW SET TO FIT RH:
           H B
           6 4
           4 2
           8 4

-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-‾-_-

M←M0
ATK←ATK0
HOFM
NPH: 3
GSIG CURRENTLY SET TO
 17 15  7  5  0 25 30 32 20 16  9  6
  3  1  8 12 18 26 27 34  2 11 33 10
 21 28 35 13  4 23 14 29 24 31 19 22

RDUP AND NDUP SET TO 1 AND 3
```

```
∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=0 ?
⎕ATK
4 3 4 1 4 3 2 1 2 1 4 4 3 1 4 4 3 1
⎕⍴ATK
18
FORM←'A(4)A(6)B(4)B(4)'
PLAN←0 4 10 14

TM=0 1 2 3
S, P, XP, F OR XF (OR ⎕) ?
UPTM (¯1↑⍋,MM) NVRT 4↑⍋,MM
⎕TM
3 2 0 ¯2
XF←1
XP←TM
S←MA

CND=⌊/TNS?  (CND,TNS,HA,H?)
⎕⍴HA
1 8

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=1 ?

TM=0 1 2
S, P, XP, F OR XF (OR ⎕) ?
¯4 EQH 0

CND=⌊/TNS?  (CND,TNS,HA,H?)
0 ¯1 MLDZ 1 3 1
TM RESET TO ¯1 4 6

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=2 ?

TM=0 1 2 3
S, P, XP, F OR XF (OR ⎕) ?
⎕MM[17;]
3 5 3 1
UPTM ⍋MM[17;]

CND=⌊/TNS?  (CND,TNS,HA,H?)
FRDGRP
     0  1  2  3  4  5
-    0  4 22 36 11  0
FRD KEPS 2
STRUC MI7
⎕⍴HA
1 8

∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘□∘
I=3 ?
```

```
TM=0
S, P, XP, F OR XF (OR ⎕) ?
⎕TGM[I-1;0]
3
UPTM 3
S←MA

CND=L/TNS?  (CND,TNS,HA,H?)
⎕⍴HA
4 8
⎕TNS
110 10 11 0
CND←⌈/TNS

∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘
I=4 ?
⎕PLAN
0 4 10 14
⎕FORM
A(4)A(6)B(4)B(4)

TM=0 1 2 3
S, P, XP, F OR XF (OR ⎕) ?
1 EQM 0
⎕
 G♯    MA7     17  8  0  3  7 10  2  5
 E     MA7     17  4  8 11  3  6 10  1
 D♯    MI7      5  3  6 10  1  5  8  0
 A     MA7     17  9  1  4  8 11  3  6
4 OVRIDC L7C

CND=L/TNS?  (CND,TNS,HA,H?)
FRDGRP
   0 1 2 3 4 5
-  0 0 4 4 1 0
FRD KEPS 4
FITMALL FN 1 3 5 7 9 11 13
TM RESET TO 5 3 1 ¯1

∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘⎕∘
I=5 ?

TM=0 1 2
S, P, XP, F OR XF (OR ⎕) ?
⎕TGM[0 1;]
3 2 0 ¯2
¯1 4 6
⎕TGM[4;]
4 3 1 ¯1
⎕SETNAMES H[0 1;]
 G♯    MA7
 E     MA7
⎕SETNAMES H[,4;]
 C♯    LG7
```

```
1 EQM 1
1 EQH 1

CND=L/TNS?  (CND,TNS,HA,H?)

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=6 ?

TM=0 1
S, P, XP, F OR XF (OR ⎕) ?
1 EQM 2
⎕TM
4 6
F←1
S←MA

CND=L/TNS?  (CND,TNS,HA,H?)

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=7 ?

TM=0
S, P, XP, F OR XF (OR ⎕) ?
¯2 OVRIDC MAC

CND=L/TNS?  (CND,TNS,HA,H?)
0 ¯1 MLDZ 1
TM RESET TO 8

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=8 ?

TM=0 1
S, P, XP, F OR XF (OR ⎕) ?
¯2 OVRIDC MAC

CND=L/TNS?  (CND,TNS,HA,H?)
0 ¯1 MLDZ 1
TM RESET TO 9 11

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=9 ?
TM=0
S, P, XP, F OR XF (OR ⎕) ?
3 OVRIDC MAC

CND=L/TNS?  (CND,TNS,HA,H?)
0 ¯1 MLDZ 1
TM RESET TO 12
⎕PLAN
0 4 10 14

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=10 ?
```

```
TM=0 1 2 3
S, P, XP, F OR XF (OR ⎕) ?
¯1 EQH 9

CND=L/TNS?  (CND,TNS,HA,H?)
0 ¯1 MLDZ 1 1 ¯3 ¯1
TM RESET TO 13 14 9 8

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=11 ?

TM=0 1 2 3
S, P, XP, F OR XF (OR ⎕) ?
¯4 EQM 10

CND=L/TNS?  (CND,TNS,HA,H?)
KEPS FRD 2

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=12 ?
TM=0 1 2
S, P, XP, F OR XF (OR ⎕) ?
¯4 EQH 11

CND=L/TNS?  (CND,TNS,HA,H?)
0 ¯1 MLDZ 1
TM RESET TO 5 6 8

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=13 ?
⎕MANDH 10 11 12
 C#    D     A     G#      :  D     MA7
 A     A#    F     E       :  A#    MA7
 F     F#    G#            :  F#    MA7
TGM[I;⍳2]←'10'

TM=10
S, P, XP, F OR XF (OR ⎕) ?
STRUC L7A3
CND=L/TNS?  (CND,TNS,HA,H?)
⎕⍴HA
6 8

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘
I=14 ?
⎕FORM
A(4)A(6)B(4)B(4)
⎕PLAN
0 4 10 14

TM=0 1 2 3
S, P, XP, F OR XF (OR ⎕) ?
¯3 EQM 10
```

```
¯3 EQH 10

CND=L/TNS?  (CND,TNS,HA,H?)

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=15 ?

TM=0 1 2 3
S, P, XP, F OR XF (OR D) ?
¯3 EQH 11
¯3 EQM 11

CND=L/TNS?  (CND,TNS,HA,H?)

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=16 ?

TM=0 1 2
S, P, XP, F OR XF (OR D) ?
¯3 EQM 12
¯3 EQH 12

CND=L/TNS?  (CND,TNS,HA,H?)

∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘O∘
I=17 ?

TM=0
S, P, XP, F OR XF (OR D) ?
¯3 EQM 13
⎕TM
7
⎕MANDH 16
 D    D÷   F      :  D÷   MA7
⎕MXC
0 1 4 11
CLASS 0
CND=L/TNS?  (CND,TNS,HA,H?)
⎕⍴HA
14 8
FRDGRP
   0 1 2 3 4 5
-  0 2 5 5 2 0
FRD KEPS 1
NAMENT
 A    -MI7   1 1
 G    -MI7   0 1
NARROW 0
OK?
TO KEEP H, ENTER: 'NAME←H', OTHERWISE CR

_¯_ _¯_ _¯_ _¯_ _¯_ _¯_ _¯_ _¯_ _¯_ _¯_ _¯_ _
```

```
ORIGINAL TARGET M:
    0 1 2 3
    0 1 2
    0 1 2 3
    0
    0 1 2 3
    0 1 2
    0 1
    0
    0 1
    0
    0 1 2 3
    0 1 2 3
    0 1 2
    0
    0 1 2 3
    0 1 2 3
    0 1 2
    0

∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘○∘

M:  0:  D#   D    C  ¯1A#
H:      G#   MA7

M:  ¯1:  B  +1E    F#
H:      E    MA7

M:  0:  D#   F    D#   C#
H:      D#   MI7

M:  0:  D#
H:      A    MA7

M:  0:  F    D#   C# ¯1B
H:      C#   LG7

M:  0:  C   F   G
H:      F    MA7

M:  0:  E    F#
H:      E    MA7

M:  0:  G#
H:      D    MA7

M:  0:  A   B
H:      C    MA7

M:  1:  C
H:      D#   MA7

M:  1:  C#   D  ¯1A    G#
H:      D    MA7
```

```
M:   0:   A     A#    F     E
H:        A#    MA7

M:   0:   F     F#    G#
H:        F#    MA7

M:   0:   A#
H:        A#    7+3

M:   0:   A#    B     F#    F
H:        B     MA7

M:   0:   F#    G     D     C#
H:        G     MA7

M:   0:   D     D#    F
H:        D#    MA7

M:   0:   G
H:        A     -MI7
```

15 Advanced Techniques and Examples

In certain idioms—and Musical Theater is one—melody and harmony can often be composed with little or no consideration given to their stylistic presentation. Sooner or later, the composer or an arranger faces the technical problem of arranging the piece, i.e., expressing the melody and harmony in a stylized manner. With a particular set of voices in mind, whether human, instrumental, or synthesized, one approaches the task with a great deal of information. Each voice has its own useful range of pitch, its own tone color, and any number of idiosyncrasies—the way tone color changes with pitch and volume, the speed of acoustic response which limits articulation, particular pitches or pitch changes that may be difficult to perform, and on and on. But we're going to refer to "voices" in a somewhat less specific way.

The same mental processes that let us "hear" melody and harmony while no one is playing them allow us to hear "voices" that are independent of instrumentation. In sensing the tonalities that constitute harmonic continuity, particular sequences of tones stand out mentally, marking the changes from one chord to another. That is, we don't "hear" the root of one chord going to the root of the next while the third, fifth, seventh, etc, similarly move to their new positions. Instead, depending on the actual structures involved, we pick out a prominent sequence of pitches (often a chromatic or "scalewise" sequence) in which the successive notes are different functions of the successive tonalities. For example, from the sequence of tonalities on the upper staff of Fig. 15-1, *voiced chords* might be extracted in a way likely to be sensed as the lower staff. Listen to the motion of each of the four voices here separately.

Even when taken as abstractions, so that these fleeting voices are unhindered by any of the physical (instrumental) realities mentioned above, there are musical forces that affect the way we assign notes to them. I'm alluding to such arcane matters as the avoidance of "parallel motion" in voices at particular intervals, the doubling of certain chordal tones, and the selection of chordal

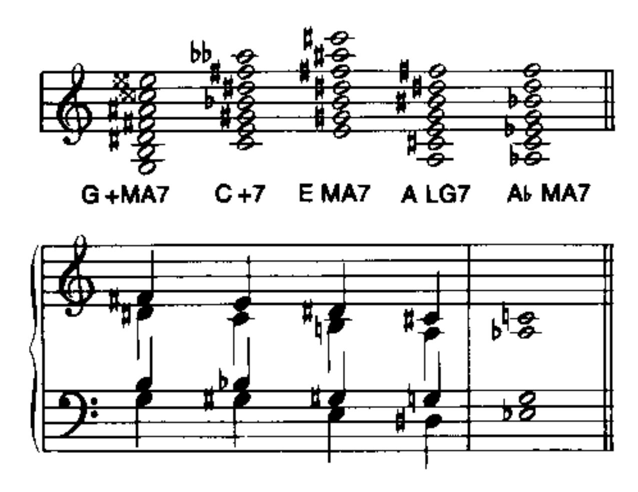


Fig. 15-1. A sequence of "sensed" tonalities for a cadence harmonizing the pitch E-flat. The lower staff shows the "sensed" voices for this cadence.

tones that can adequately support a particular melodic fragment. At this introductory level, the reasons for such requirements can remain obscure. It may take some ingenuity to program such requirements into an overall procedure, but quite simple programs that ignore these matters can be used by beginners. As sensitivity increases, one can begin to hypothesize about and experiment with voicing and orchestration. Never before have such facilities been available to study such questions, and I would most emphatically encourage anyone with a computer and sound generating hardware to do so.

In what follows, I will try to explain some approaches to these problems. However, I will not shown my program listings because of some extrinsic complications: The programs made extensive use of special device drivers for particular display and printer hardware.

VOICING AND VOICE LEADING

A convenient first step, after melody and harmony have been composed, assigns chordal tones to abstract voices. With no other embellishment, this produces accompaniment to melody in the form of "sustained" harmony. This is the form used in most of the above examples where all voices attack simultaneously and sustain their individual tones for the full duration of the chord. When more interesting motions are to appear in the accompaniment, the sustained form that is the product of this first step will provide an excellent guide for these motions. So it is well worth examining the logic needed for this first step.

The *voicing* of a particular chord relates to the order in which chordal tones are assigned to the consecutive voices, from lowest voice to highest. *Voice leading* refers to the way each voice moves from its assigned pitch in one chord to its pitch in the next. The terminology rests on the perception that each musical line or voice has a life of its own that can be seen in "snapshots" as well as in "movies." The logic of any voicing algorithm must keep track of two sets of indices—one to associate with the position of each voice or musical line and the other to assign pitches or chordal tones to these voices. For example, the vector 3 7 1 5 could be used to assign the third of the chord to the lowest voice (index equals zero), the seventh of the chord to the next higher voice (index of one), and so on.

Someone more at home in the programming world than in the musical one might express the same voicing by the vector, 2 4 1 3. Again the chordal positions for each voice are implicitly taken as 0 1 2 3, from lowest to highest. But now the chordal functions (the positions in the harmonic structure from which the chord is extracted) are related to column indices of the H vector, the struc-

ture chosen in the HOFM algorithm for each harmonization. Because H[I;0] contains the structure index (pointing to a row of the SIGMA array), the entries in this vector single out the second, fourth, first, and third tones in the harmonic structure directly, as if ignoring the index origin. These indices can be related to the musical ones simply by indexing the odd integers:

```
odd integer 1 3 5 7 · · ·
index       1 2 3 4 · · ·
```

so that 2 4 1 3 in the programmer's eyes corresponds to 3 7 1 5 in the musician's.

Regardless of the form of indexing that is chosen, the real task of a voicing algorithm is to drive the voices collectively in a way that displays the tonality sequence while maintaining musical independence of the individual lines. The sequence of tonalities can be faithfully expressed rather mechanically; for example, if we are using four voices for accompaniment, the four lowest notes in each structure can be selected. Keeping the voices independent implies that we do *not* assign particular notes to particular voices according to any obvious, mechanical scheme. That certainly rules out "constant parallel" relationships which would have the Nth voice always play the Mth note of the current structure.

Logically, all possible voicings of a given chord as well as all possible voice leadings between two given chords can be characterized by permutations. Obviously, 3 7 1 5 is a permutation of the four lowest harmonic functions, 1 3 5 7. If the next chord is voiced in functional order, 1 5 7 3, that too is a permutation of the first four functions, and it is also a permutation of the previous voicing:

```
1 5 7 3 = 1 3 5 7 [0 2 3 1]
1 5 7 3 = 3 7 1 5 [2 3 1 0]
```

Schillinger pointed this out and suggested methods that appeal to a programmer's instincts. Despite his prescience in such matters, he had to base his thoughts on the reasoning of his day. Then, arguments in concise mathematical form carried credibility. Whether or not they were practical, they faced no challenge from logical algorithms that were too exhausting to be carried out without error.

Today, an initial method based on permutations could be embellished by controlling the permutations through preference arrays which take into account the root-tone motion between two chords, the harmonic functions present in each chord, the freedom category of the change in tonality involved, and so on. While it may be more difficult to program, an algorithm founded on musical logic rather than permutations will be easier for a composer to control and will lead to far more satisfying musical results.

OUTLINE OF A VOICING ALGORITHM

An initial chord must be specified in order to prime our voicing logic. It need not be the first chord in the harmonic continuity because the logic of voice leading will establish the next chord, whether "next" applies to the preceding or following one. Also, the word *chord* is being used here to specify a particular set of pitches extracted from a given harmonic structure and placed in a particular (voiced) order. For simplicity, say we are going to use four-part voicing, and our initial chord is to have the harmonic functions third, tonic, fifth, and seventh (3 1 5 7) reading from the lowest voice upward. To complete the specification, say we choose A♭ as the root tone and 17 as the row index from our structure array, SIGMA. This will give us the vector

```
P1←0 8 3 7.
```

Notice, by saying these are to be "read upwards," we can use pitch values that don't take octaves into account; the lowest pitch is zero (C), the next higher pitch is eight (A♭), the voice immediately above that plays the pitch three (E♭), and the highest voice plays seven (G).

The next chord is to be voiced by our algorithm, and we will say the pitches are to be selected from the four lowest harmonic functions (1 3 5 7) of the given harmonic structure. Let's

assume the next structure in the given composition is in the LG7 class (row 16 of SIGMA) and is to be built on D♭ (pitch number one). The algorithm can then construct the pitch vector

```
V ← +\1,SIGMA[16;⍳3]
```

which will be 1 5 8 11. This vector contains the notes that are to be in the chord, but those must now be assigned to the four voices so that the voice leading between the initial chord and this one is musically acceptable.

An array of interpitch distances can now be created by the algorithm, showing how far each note in P1 must move to reach every note in V. In principle, the "jot-dot-minus" operation will do this:

```
D ← V∘.−P1
```

In practice, however, we want to adjust the resulting values to represent the shortest distances, up or down:

```
D←D − (×D) × TONSYS × (⌊|TONSYS÷2)
< D.
```

A single column of this array (**D[;J]**) shows how far one particular voice (the Jth voice) in the chord P1 would need to move in order to reach every note in the chord that is not yet voiced. Of course, each voice can be given only one note in the new chord, and so the remainder of the algorithm must choose the "best" entry in each column. To do this in the typical **HOFM** fashion requires a collection of determining conditions that have default settings which may be altered interactively by changing one or two parameters.

Algorithmic Conditions For Best Voicing

The conditions I use and their default modes work as follows. First, the direction of the melodic motion in going from the last pitch of the earlier segment of melody to the first pitch of the later one is found. Every column of the D-array then has its elements reordered using grade-up order for motions opposite to the melody, followed by grade-up order for parallel motions. In general then, the smallest motion for each voice moving in the direction opposite to the melody is placed first in each column, and the largest voice movement in the same direction as the melody is placed last. A separate subcondition decides whether a zero motion should come first (the default), come between the positive and negative intervals, or go last. If the intervals in one column were initially (¯2 1 5 ¯4) and the melody moved downward, this column would be reordered to (1 5 ¯2 ¯4). Because the *rows* of D correspond to consecutive harmonic functions, in APL we find

```
(1 3 5 7)[¯2 1 5 ¯4⍳ 1 5 ¯2 ¯4]
```

equals 3 5 1 7, indicating that the voice represented by this column would by preference play the third of the "next" chord. If for some reason that pitch is ruled out, the remaining choices for this voice in order of preference are the fifth, tonic, and seventh.

The next condition controls the order in which voices (columns) are to be assigned pitches. By default, I work from highest to lowest voice. Whatever order is used, other conditions (described below) are applied to find the most preferred pitch (the "best" row in the given column) that meets all requirements. If, say, the first and second voices have been assigned satisfactory tones, and then it turns out that the conditions can't be met for the third voice, the algorithm returns to the second voice and changes the selection to the next preferred pitch according to the conditions for that voice. It then goes back and tries to fill the third voice again. It may turn out that no setting of the second voice alleviates the problem, so the first voice needs to be reset before the chord can be completely specified.

The highest voice need only avoid crossing (going higher than) the lowest pitch in the melodic segment. This can be overridden to make the limit any higher pitch, whether in the melody or not. Notice that this implies that the melody uses relative octave indications here, so the highest voice in the *initial* chord is taken to lie within one octave of and

below the lowest pitch in that initial melodic segment. The algorithm then can keep track of the positions of all the notes (in all the voices) in the later chords.

The first (highest) voice, as well as the others, must not duplicate the two melodic tones connecting two melodic segments i.e., if a voice plays the same pitch as the last note in its melodic segment, it must not move to the same note as the first pitch of the following segment. An additional rule prohibits duplication of the first note in a melodic segment if the motion to that note is in the same direction as the melodic motion. For example, if the melody goes from E in one segment upward to G in the next, no chordal voice should move to G if the motion is upward.

The later (lower) voices undergo additional scrutiny. A later voice may not ordinarily duplicate an earlier one in the same stratum. In addition, voice cannot cross; the lowest voice must remain the lowest, and similarly, the other voices must stay in their relative positions. Selecting voices in the default order from highest to lowest works most efficiently because only the position of the previous voice, beginning with the melody, needs to be checked. This sequence also seems to favor the production of "range-balanced" chords—chords in which the closer intervals are in the upper voices. But better control is explicitly provided by some vectors which will be described below.

A final condition may (by default) be applied to a "completed" chord. If the tension within the resulting structure exceeds 1,000, a further test is made of the "offending" voices—those whose pitch names differ by one semitone. If their separation is greater than one octave, they interchange pitch names but stay in their "octave," i.e., each moves toward the other through one semitone so that minor ninths are converted to major sevenths. If this violates any other condition, the chord is considered unacceptable and the voice-selection process resumes until a satisfactory pitch assemblage is found.

Other Controls for the Algorithm

By default, the algorithm expects to use one stratum containing as many voices as are in the initial chord. The harmonic functions for these voices will be taken to match the relationships between the initial chord and the initial structure. However, the STRATA vector can be explicitly set, for example, to 1 3 2, indicating there are to be three strata, the lowest pitch in the initial chord is (the only one in) the lowest stratum, the next three pitches comprise the middle stratum, and the two highest pitches from the top stratum. In this case there will be three harmonic function vectors established: HF1, HF2, and HF3. By default they are set to correspond to the functional relations between the initial chord and the initial structure for each stratum. Of course, they can be reset by the user at any time, keeping a number of rules in mind.

When the number of voices in a stratum is greater than the number of entries explicitly set by the user in the harmonic function vector (HF*n*) for that stratum, the vector is extended by cycling over its entries as required to match the number of voices in the stratum. For five voices with three entries in its HF vector (1 3 5), the vector would become (1 3 5 1 3). In the reverse situation, where a stratum contains fewer voices than allowed harmonic functions, another set of rules applies. If there are N voices in such a stratum, the first (N-1) functions in its HF vector must be used.

When any one of the remaining functions in the given vector appears in a voice, the other "extra" functions become invalid for that chord. For example, if we have four voices and the function vector (1 7 11 3 5 9), possible chords will all contain the root, seventh, and eleventh, while either the third, fifth, or ninth of the harmonic structure will also appear. However, even the "N-1 rule" can be overridden by including a minus sign before an entry that comes before the (N-1)th. That is, (1 7 ⁻11 3 5 9) would force only functions one and seven to appear in the chord along with, for four voices, any two of the functions, 11, 3, 5, and 9. A function vector that begins with a negative value (⁻1 7 11 3 5 9) provides the most freedom; for a tetrad, any set of four unique entries could be chosen from this vector.

Different strata can be given individual direc-

tional trends and range limits. I have experimented with a number of ways of doing this, but there are so many different possibilities that I haven't been able to gather them all into a single scheme that is easy to use. Further, I frequently find it hard to decide whether this should be treated as a problem in voicing or in stylistic accompaniment, which we'll discuss below. The most efficient approach so far relies on another (growing) library of auxiliary subroutines. Instead of discussing my current (and tentative) methods here, I'll just point out some of the problems they address.

SOME UNRESOLVED VOICING PROBLEMS

The following problems seem to be problems of voicing, and it would be desirable to have voicing algorithms that solve them. So far however, I find it easier to treat them interactively as problems of stylistic accompaniment, as we'll see below.

The highest or lowest voice (or both) in a stratum may "need" to be confined to a specific range. While voices in one stratum may not cross, voices in adjacent strata may do so if overlapping range specifications are set. Range specifications are quite tricky here because my voicing routines use pitch values in the "reference octave"; only the melodic line carries octave information in my current programs.

Range and octave considerations can also affect the "openness" or the "balance" of a chordal setting and require explicit control. *Openness* refers to the intervals between adjacent voices in a chord; in close harmony, the pitches are distributed with minimum separations between voices. The tetrads (C E G B), (E G B C), (G B C E), etc. have this "close" property. The form (C G E B) is an open one and so is (C E B G), but the latter one may not be "balanced." If the upper voices are more widely separated than the lower ones, the chord is said to be unbalanced or "top-heavy" and it sounds "weak" (there is a loss of resonance in the sound).

Further, in a low register, the low frequencies of close voices interfere to create an effect often characterized as "muddy." If the lowest interval in the last chord (from C up to E) is a tenth rather than a third—i.e., an octave plus a third—the chord will have better balance and resonance. In any case, after the harmonic continuity has been completely established, each voice plays a completely "correct" musical line, and so the positions of the various voices can be interchanged, e.g., the two lowest voices might be interchanged so that the sequence of voices, from lowest pitch to highest, would be 1 0 2 3 (for four-part voicing). Questions of balance must then be set aside because they depend on orchestration rather than voicing.

Given a melodic segment and its accompanying harmonic structure, the choice of alternative harmonic functions for voicing the chord is often related to the melodic content. That is, when I see a melodic line and its *named* harmony (root and class), I somehow let the notes in the melody determine which harmonic functions should provide chordal support. The use of complete seven-part structures for melody and harmony allows all the melodic tones to be considered as harmonic functions. While it's easy to tell what functions of the harmony are "missing" in the melody, it is not at all obvious to see what functions should be in the chord to support the melody in such a way that both, melody and chord, express the proper tonality with satisfactory resonance. Functions that appear in the melody may or may not produce an adequate effect if duplicated in the chord, depending on their duration and position in the meter, as well as on the functions themselves. This is another area where a good hypothesis is needed to convert a musician's understanding to logical programs that will simulate his methods.

Another facet of voicing involves "chromaticization" of harmony. This is the sort of thing shown in Fig. 9-13 without computer assistance. After the sustained voiced has been worked out, individual parts are allowed to move chromatically and scalewise between and around their specified tones instead of sustaining them. The intermediate structures produced in combination with the melody should display tension values within controlled limits. The voices need not reach their "target" tones simultaneously, nor need they hold those pitches while lagging voices catch up. The

original chord then serves only as a template for what occurs without ever necessarily appearing intact in the harmonic continuity.

A PROGRAM FOR CREATING STYLIZED ACCOMPANIMENT

Given melody, harmony, and voiced chords, our ultimate goal is to produce something resembling a musical score. While there are countless problems of musical logic to be solved in this process, before any of them can even be considered, a practical form of representation must be worked out, and this is the biggest problem of all!

The program to be demonstrated here, unlike the previous interactive programs shown, was developed explicitly to use a video display screen as the primary device for interaction rather than a typewriter-like keyboard/printer. This means that the user can indeed work with a "score," pointing or moving the cursor to any part of the screen and making entries or changes at will. When the return key is pressed, information anywhere on the screen can be examined and used by the program. No longer must the user enter complete statements such as "1←9." He can now point to the area on the screen that contains the current segment number and enter the value he wants to use. While he no longer needs to think in terms of a computing language, he must understand how to communicate with the program being used. It would be ideal if all such communication could be carried out through natural language, but words just are not efficient enough for such unnatural subjects as music.

The details of a specific form of representation for the rythms in the various lines of a score and for the instrumental content of such a line (a single instrument, a section of an orchestra, a pianist's left hand, etc.) remain complex, no matter what kind of interactive devices are to be used. I'll describe a notation system here which was worked out so laboriously and subjectively that it should only be viewed as a suggestion for approaching the problem of representation. If a standard form of representation is ever worked out, it may have nothing at all in common with the one to be described.

The pieces shown in "The Cybernetic Song Book" following this chapter were arranged by the program to be described here. That program accepts as input the results of the HOFM program treated according to the voicing algorithm described above. As you might expect, a session is broken into segments that correspond to those used in HOFM. The samples shown in Figs. 15-2 and 15-6 are "printer snapshots" of the display for some of the segments used to develop the second piece in the song book. The "I-time" message appears in red at the top of the screen, information displayed by the program is printed in blue, and user entries are in green. (The blue and green are far easier to distinguish on the display than on the four-color printer!) Each segment of a piece occupies a separate display screen and so, although printer paper (according to my dealer) can be bought "for a song," printer snapshots quickly deplete my paper budget.

The program begins each segment by clearing the screen and displaying the segment number in red:

```
SEGMENT # 14
```

It then waits for the user to change the segment number or accept it. When the return key is pressed, the program shows the target melody and (voiced) harmony for the segment whose number is on the screen.

```
TM = 0 0 2 4 0 4 2
H = A# LG7 (1 5 7 3)
R: □ (blinking cursor)
```

The program then waits for a rhythm to be entered.

RHYTHMIC REPRESENTATION

Two symbols are used to describe rhythms, one to indicate melodic attacks and the other to "mark time" for the expected accompaniment. If the target melody for the segment has five notes, the specification shown

```
SEGMENT # 0                                SEGMENT # 3

TM = ¯5 ¯4 ¯1 0 3                          TM = 5 3
 H = SKIP                                   H = C MI7 (3 1 5 7)

R:    00000                                R:    LIKE 1

M(FN):  0    0    0    0    0              M(FN):  11   .    .    3

                                           V1           .    .    4    .
                                           V2           1    2    3    2
D:      G    G#   B    C    D#
                                           D:      F    .    .    D#
                                                   .    .    A#   .
SEGMENT # 1                                        D#   C    G    C

TM = 8 7
 H = D -MI7 (3 1 5 7)                      SEGMENT # 4

R:    0..0                                 TM = ¯1 0 3 8
                                            H = F LG7 (1 7 3 5)
M(FN):  5    .    .    11
                                           R:    0000
V1           .    .    4    .
V2           1    2    3    2             M(FN):  11   5    7    9

D:      G#   .    .    G                   LIKE 1  .    .    4    .
        .    .    C    .                           1    2    3    2
        F    D    G#   D
                                           D:      B    C    D#   G#
                                                   .    .    C    .
                                                   F    D#   A    D#
SEGMENT # 2

TM = 1 2
 H = G LG7 (1 5 7 3)

R:    LIKE 1

M(FN):  11   .    .    5

V1      1    3    4-1  4

D:      C#   .    .    D
        G    F    A#   B
```

Fig. 15-2. "Snapshots" of the first few screens of the interactive session that developed the second piece in "The Cybernetic Song Book."

. . ● . . ● . ● ● . ● . . . . .

might imply the rhythms shown in Fig. 15-3. Here, the dots are regular enough to represent the ticking of a metronome. But the same representation could also account for the rhythm of an accompaniment in which dots keep a less regular but quite specific beat, as in Fig. 15-4.

Although the choice of symbols can be arbitrary, the program must know what pair of symbols to expect. Blanks cannot be used as separators and should not be used as one of the symbols, so
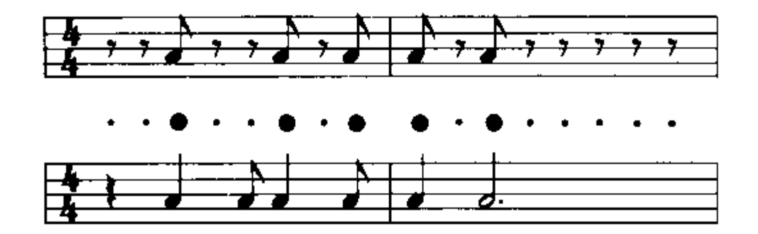


Fig. 15-3. Two possible metronomic interpretations of a "dot-rhythm."

that the entire rhythmic pattern is explicitly visible. The rhythm

● ● ● ● ●

implies five attacks with no special indications to describe or leave room in the score for the meter or the accompaniment.

The rhythm for a segment can be made identical to that for a previously treated segment by a LIKE specification. That is,

LIKE 3

would be equivalent to reentering the same rhythmic notation that was used in segment number three. If the LIKE-segment number given by the user has not yet been defined, the last rhythm that was used is taken for the current one.

THE SCORE

After verifying that the rhythm entered accounts properly for the melodic attacks (the number of entries in TM), the program redisplays the melody in a form that spells out its harmonic and rhythmic content. For example, if the above rhythm

. . ● . . ● . ● ● . ● . . . . .

were applied to the five melodic tones (C D E F G) harmonized by an F-major tonality, the following line would be printed:

M(FN): . . 5 . . 13 . 7 1 . 9 . . . . .

The "M(FN)" indicates that what follows shows the melody in functional form, i.e., each note is given in terms of its "functional" position in the harmonic structure. The rhythm here was defined by a string of 16 characters. This line, beginning with M(FN), rewrites the melody, using an expansion factor or four, in 64 character positions. The user is now to consider this line to be a temporal template, entering directly below it lines of characters that describe voices or groups of voices. The characters so entered must line up with the template so that rhythmic simultaneity is directly observable. The space allotted to each "time unit" is now large enough (four characters) to contain any of the allowed codes (to be described below) for indicating voice behavior. The sequence

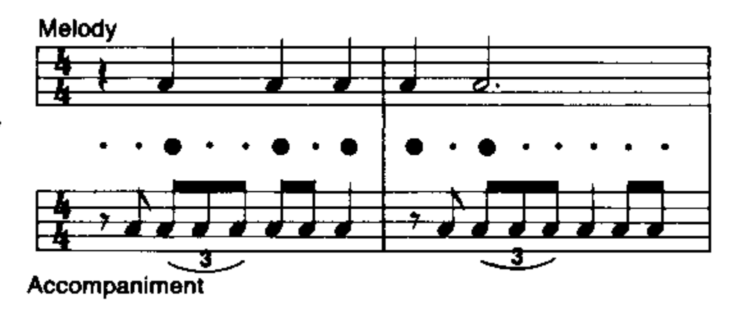


Fig. 15-4. The same dot-rhythm of Fig. 15-3, interpreted to mark the attacks of the accompaniment.

```
M(FN): . . 5 . . 13 . 7 1 . · · ·
V1:    . 4 . 2 4  . . . . 4 · · ·
V2:    . 3 . 2 3  . . . . 3 · · ·
V3:    1 . . . . . 2 . 1 . · · ·
```

describes a rhythmic setting for four voices, including the melody. Translating the melodic functions to notes in an F-major tonality and the voice entries into chordal indices where 1, 2, 3, and 4 mark the positions of the tones F, C, A, and E (in the H array), we have the score shown in Fig. 15-5.

After pressing the Enter key, the "score" is reprinted (following a quad symbol) with pitch names substituted for functions and chord indices. If nothing had been entered below the **M(FN)** line, the program would check to see whether LIKE had been used to define the rhythm. If it had not been used, just the melody would be displayed after the quad symbol. But if LIKE had been used, the coded score for the indicated LIKE segment would appear below the melodic template, and the program would return control to the user so that he might edit it or accept it without change. After the user strikes the Enter key, whatever is then in this area of the screen would be decoded and printed as the "score" following the quad symbol.

The maximum number of instrumental lines can be given beforehand, so that the program can display an "empty" (i.e., dotted) form for the score. The user then can replace particular dots with the required characters with less effort than would be needed if he assumes total responsibility for rhythmic alignment with the melody. Incidentally, labels such as "V1" and "VC" are entered by the user for his own convenience. They carry no meaning to the algorithm.

HIGHLIGHTS OF THE SAMPLE SESSION

Figure 15-2 begins with "**SEGMENT # 0**." The target melody was left unharmonized for this segment, and so the "chord" is given as **SKIP**. When the melody is restated in terms of "harmonic functions," only the rhythmic notation can be shown (there being no structure to refer to). No accompaniment was described in the next (blank) line, so the quad-score just contains the melody, the five unaccompanied notes that begin the second piece in the song book.

Segment number one shows the more usual situation. Notice, the chord is described by root (D), class (**⁻MI7**), and voiced functions (3 1 5 7). The tones in the target melody (eight and seven) are shown to be the fifth and eleventh in the given harmonic structure. An accompaniment containing two parts, V1 and V2, was defined next. The numbers used here refer to positions in the above chord:

functions in chord	3	1	5	7
pitches in D – MI7	F	D	G♯	C
positions in chord	1	2	3	4

The score following the quad symbol then translates the melody and these two accompanying parts to pitch names.

For the next segment, "LIKE 1" put the

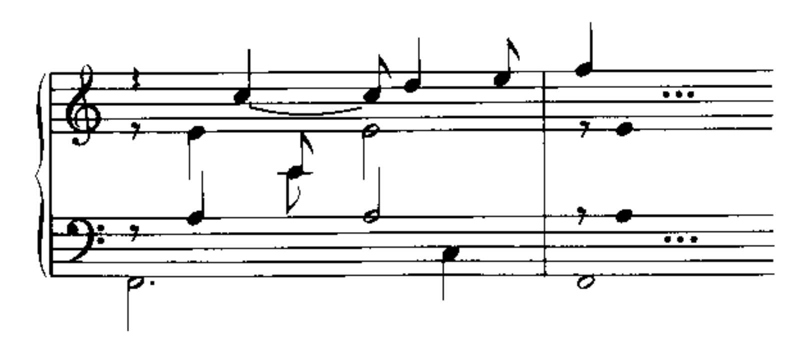


Fig. 15-5. The musical score derived from a set of coded voices.

```
SEGMENT # 18

TM = 13 11 10 11 13 11
 H = B MA7 (5 7 1 3)

R:     000000

M(FN):   9    1    7    1    9    1

V1       .    .    M-2  .    .    .
VC       D    .    .    .    .    .
V2       .    .    F1   .    .    .

D:       C#   B    A#   B    C#   B
         .    .    F#   .    .    .
         D#   .    .    .    .    .
         B    .    .    .    .    .
         A#   .    .    .    .    .
         F#   .    .    .    .    .
         .    .    B    .    .    .
```

```
SEGMENT # 19

TM = 13 11 10 11 13 11
 H = E LG7 (1 3 5 7)

R:    LIKE 19

M(FN):   13   5    11   5    13   5

V1       .    .    M-2  .    .    .
VC       D    .    .    .    .    .
V2       .    .    F5   .    .    .

D:       C#   B    A#   B    C#   B
         .    .    F#   .    .    .
         D    .    .    .    .    .
         B    .    .    .    .    .
         G#   .    .    .    .    .
         E    .    .    .    .    .
         .    .    B    .    .    .
```

```
SEGMENT # 20

TM = 13 11 10 11 13 11
 H = D MA7 (1 3 5 7)

R:    LIKE 19

M(FN):   7    13   13   13   7    13

V1       .    .    M-2  .    .    .
VC       D    .    .    .    .    .
V2       .    .    F1   .    .    .

D:       C#   B    A#   B    C#   B
         .    .    F#   .    .    .
         C#   .    .    .    .    .
         A    .    .    .    .    .
         F#   .    .    .    .    .
         D    .    .    .    .    .
         .    .    D    .    .    .
```

```
SEGMENT # 21

TM = 16
 H = C# LG7 (3 5 7 1)

R:     0.....

M(FN)    9    .    .    .    .    .

VM       M5Δ  .    .    .    .    .
VC       D    .    .    .    .    .
V1       .    .    F1   .    .    .

D:       E    .    .    .    .    .
         B    .    .    .    .    .
         G#   .    .    .    .    .
         E    .    .    .    .    .
         C#   .    .    .    .    .
         B    .    .    .    .    .
         G#   .    .    .    .    .
         F    .    .    .    .    .
         .    .    C#   .    .    .
```

```
SEGMENT # 22

TM = 11   9    8    9    11   9
 H = A MI (5 7 1 3)

R:     000000

M(FN):   9    1    7    1    9    1

V1       .    .    M-1  .    .    .
VC       Δ-2  .    .    .    .    .

D:       B    A    G#   A    B    A
         .    .    F#   .    .    .
         C    .    .    .    .    .
         A    .    .    .    .    .
         E    .    .    .    .    .
```

```
SEGMENT # 23

TM = 11   9    8    9    11   9
 H = D MI (3 5 7 1)

R:    LIKE 23

M(FN):   13   5    11   5    13   5

V1       .    .    M-1  .    .    .
VC       Δ-3  .    .    .    .    .

D:       B    A    G#   A    B    A
         .    .    F    .    .    .
         D    .    .    .    .    .
         A    .    .    .    .    .
         F    .    .    .    .    .
```

Fig. 15-6. Snapshots of some screens used in developing the "B" theme for the second piece in "The Cybernetic Song Book."

melody in the same rhythmic form as for the previous segment. I simply pressed the Return key, and the two parts, V1 and V2, from the previous segment were copied directly as

```
V1    . . 4 .
V2    1 2 3 2
```

But I could now see something that I have not yet been able to get a program to recognize: The melody contains one of the chordal tones (the fifth) in a particularly uncomfortable way for the planned accompaniment. So the lower voice, V2, was deleted with the press of a button, and the remaining voice was changed to

```
V1       1  3  4-1 4
```

Because the chord has the functional sequence (1 5 7 3), V1 refers to chordal tone number one (function one, the root), tone number three (the seventh), a "special" entry which I'll explain shortly, and chordal entry number four (the third). That special entry in the third position is decoded by the algorithm to mean the tone that is one scale step below the fourth chordal voice. The scale referred to here is the one derived from the seven-part harmonic structure containing the chord. An entry such as **4 + 2** would call for the note two scale steps above the fourth chordal voice. In a similar way, the decimal notation, **4 – .1**, would refer to the pitch one chromatic step below the fourth chordal tone, regardless of the harmonic structure.

Segments three and four did not have this conflict between the melody and planned accompaniment, so that the setting from segment number one was directly applicable to both. But notice in segment four, the rhythm was defined explicitly to account for four attacks (unlike segment one), and "LIKE 1, <carriage return>" was entered in the accompaniment area. This brought in the proper chordal indices, but left the LIKE indication on the screen instead of replacing it with the labels, V1 and V2.

In Fig. 15-6, some of the other special codes can be seen in the coded score. Segment number 18 corresponds to the tenth measure of the piece, where the time signature changes to 3/4. The first line of the accompaniment here contains the code M-2. An *M-code* refers to the melodic pitch that happens to be in effect at the moment this code appears. M plus or minus an integer or decimal value singles out the current melodic tone to serve as a reference point for a scalar or chromatic displacement, e.g., M-2 means two scale steps below the melodic pitch. The second accompaniment line (labeled VC) uses the quad symbol, indicating that the four lowest chordal tones are to be played simultaneously here. In the next line, for the part labeled V2, a specific harmonic function is called for through the use of an *F-code*. F1 implies the root is to be played at this point in this part.

Similar treatments are given the next two segments (19 and 20). There are only two features that might be novel here. First, the use of LIKE with the current segment number (19) refers to the previously defined segment, whether in sequence or not; when concentrating on musical matters, even subtracting one from an integer can be so great a distraction that I sometimes use the current segment number appearing on the screen. And second, "F1" was changed to "F5" in the third part (V2) to specify the fifth of the harmonic structure.

In the remaining segments shown, the delta symbol appears in two different ways. Just as the quad symbol implies four, a delta implies the three lowest tones of the chord. Alternative forms such as "**Δ – 2**" or "**□ – 3**" designate three or four notes taken sequentially from the voiced chord, but skipping the "subtracted" voice; thus, "**Δ – 2**" implies a triad using voices 1, 3, and 4. The form **M5Δ** says to use an ordinary triad—i.e., one built in thirds—constructed from the harmonic scale in which the melodic tone is the fifth and highest note. For example, with a tonality that uses the seven white notes, regardless of the root, if the melodic tone were A and the fifth of this triad, the third and root of the triad would be F and D, respectively. The three voices indicated by **M5Δ** would then be, from lowest to highest, (D F A). In the case of segment 21, don't be confused by what appears to be an E-
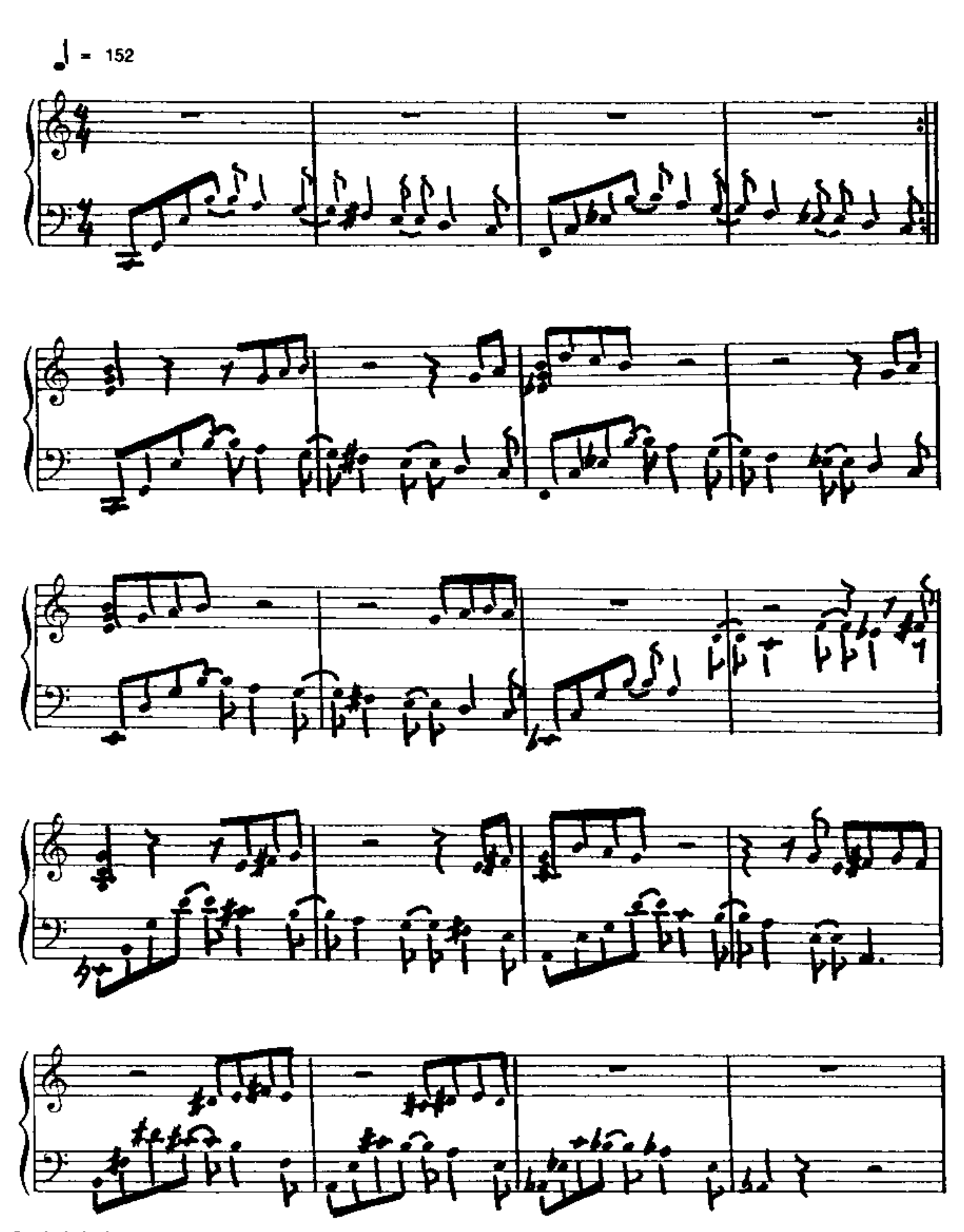
major triad with the root as the highest tone. For the given harmonic structure, C# LG7 (row 16 of SIGMA), the algorithm was clever enough to recognize the melodic function, 9, as the second scale step, but was unable to address the triad as (G# B D##).
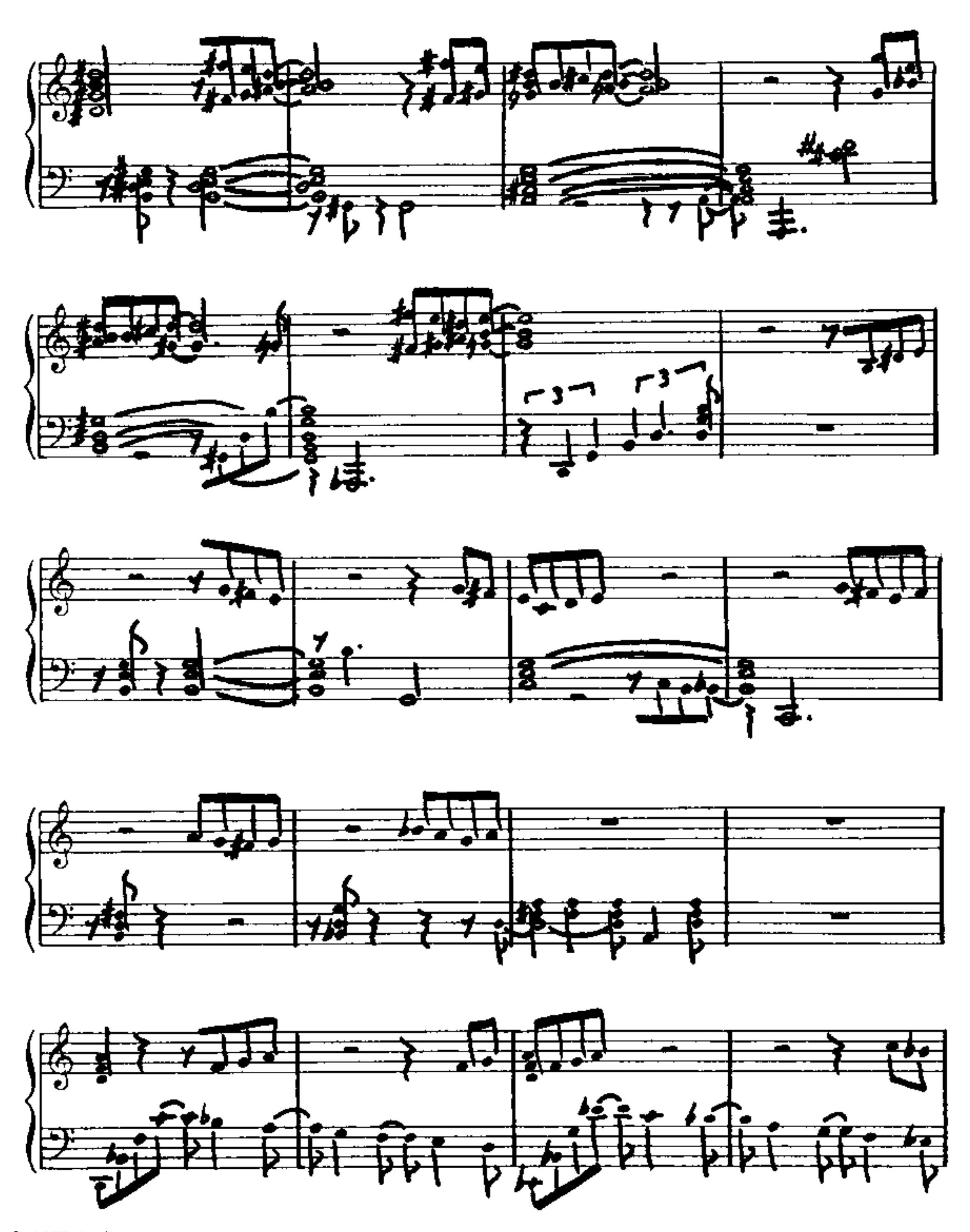
The Cybernetic Song Book
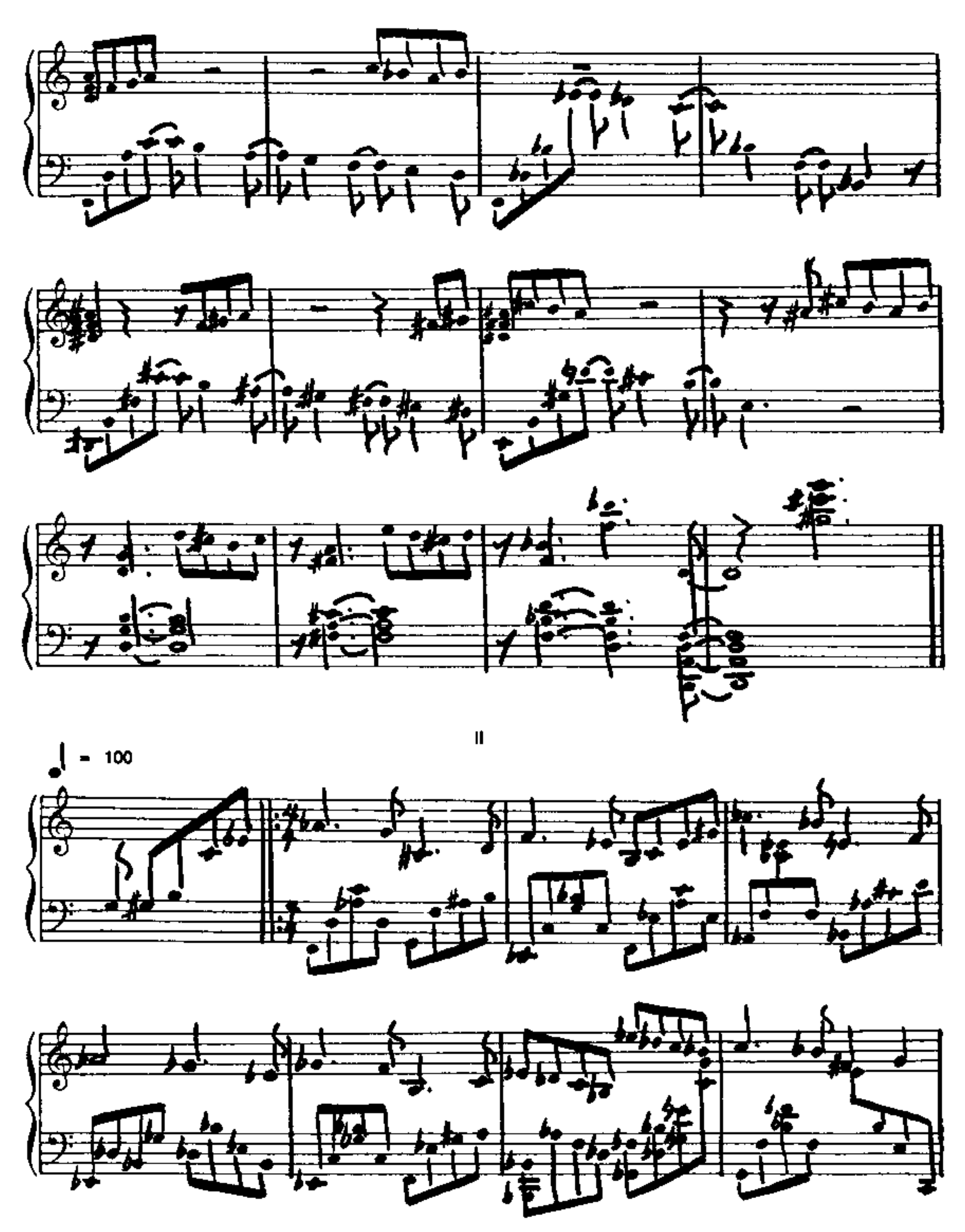
The following pieces were selected from a set of songs that grew out of an experiment tinged with a shameful trace of deception. My purpose was to show that "computer music" need not speak with a technological accent. The laboratory for the experiment was a workshop for writers for the musical theater. All the songs were programmed to fit words composed by lyricists or, when desperate for material, by me.
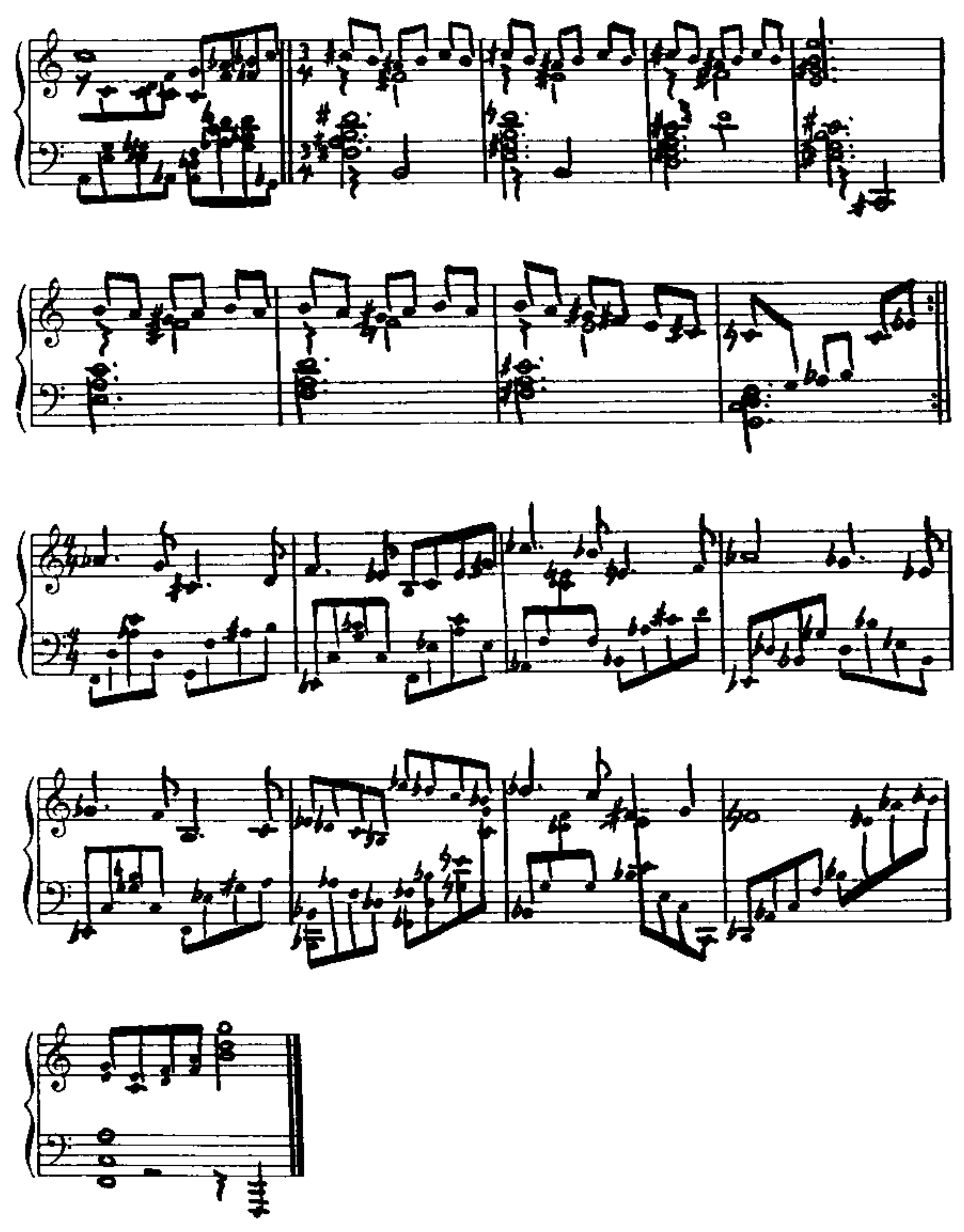
It would be fair to say that most of the lyricists did not view technology as amicable or even benign, and I valued their friendship and their personal feelings too highly to reveal that they had in some way "collaborated" with a machine. In fact, one writer, who knew of my work in this area, pleaded with me to tell him that the music for his song, which he liked so much, was my own creation. I felt nothing reprehensible about assuring him that I was indeed the composer.
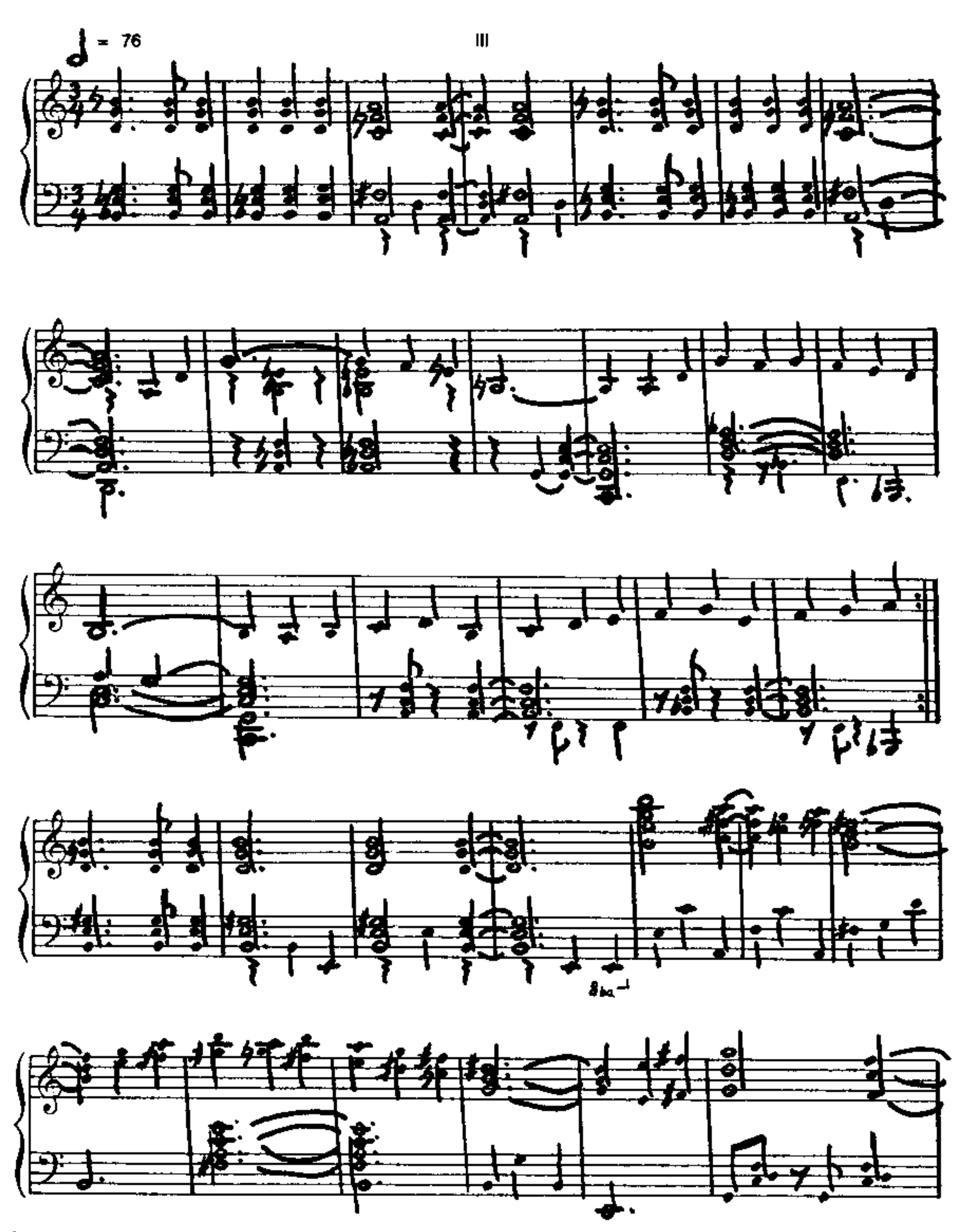
As for the director of the workshop, his outspoken views were such that I didn't dare bring up the subject with him. From all the comments on the music, assuming no one was being reciprocally sensitive to my feelings, the experiment was a success. However, as a "control run," I demonstrated the songs to various people not associated with the workshop and explained beforehand that I had used a computer as a tool in composition. Apparently the serum of truth can act as an anesthetic on the esthetic judgement of some listeners! While no one expressed dislike, some were too disconcerted to be able to express an opinion, and others found the whole thing "interesting."


♩ = 152
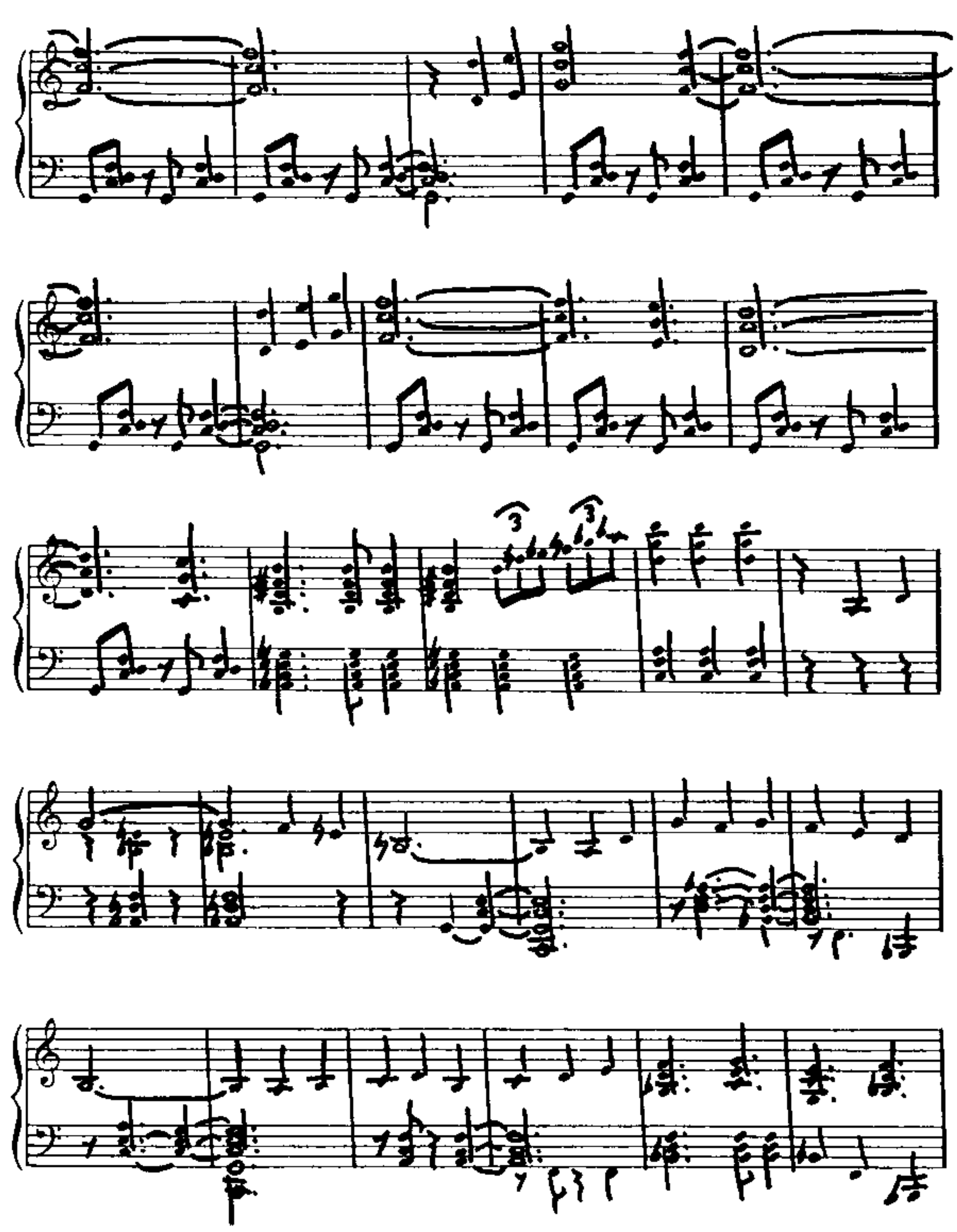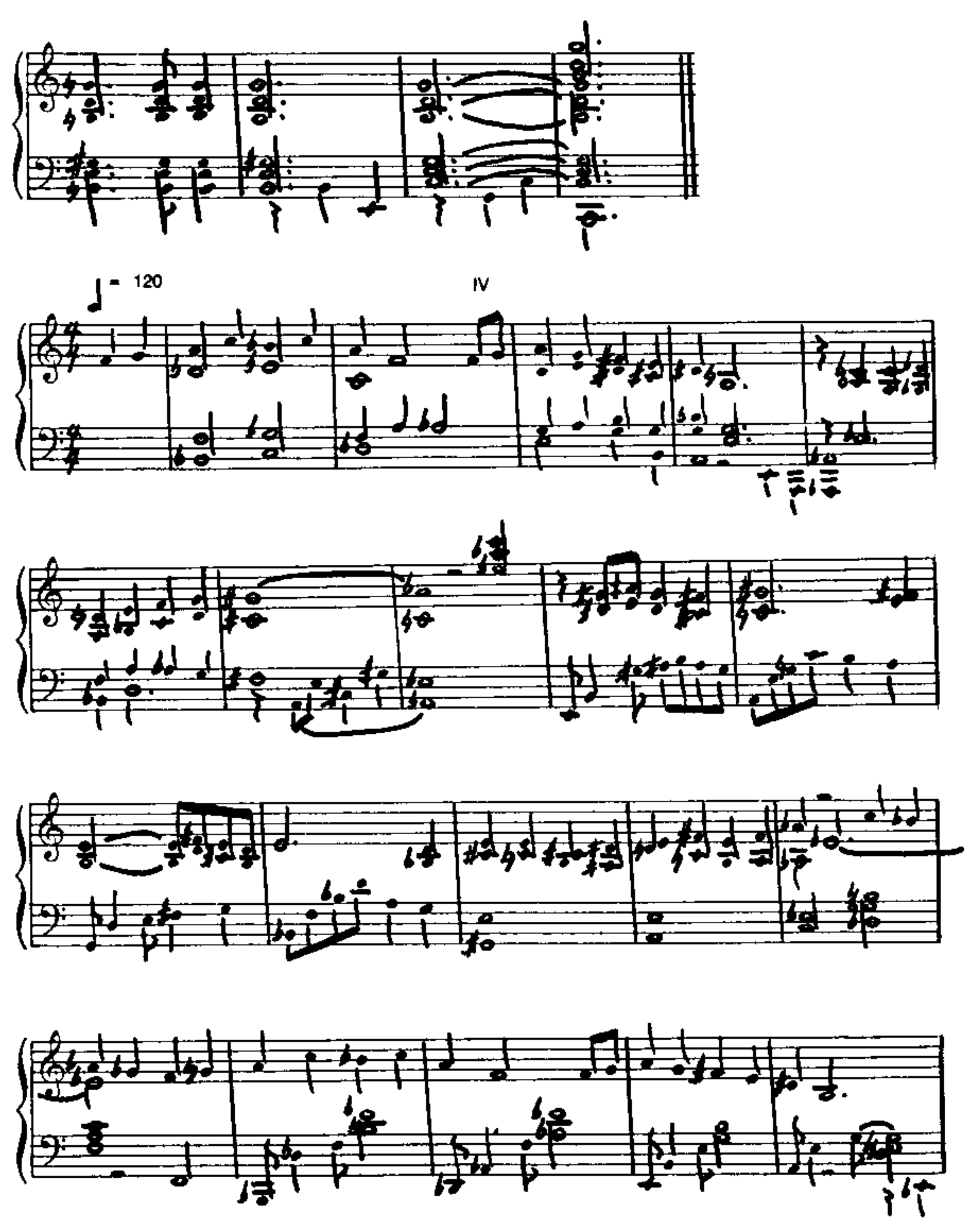



II
♩ = 100
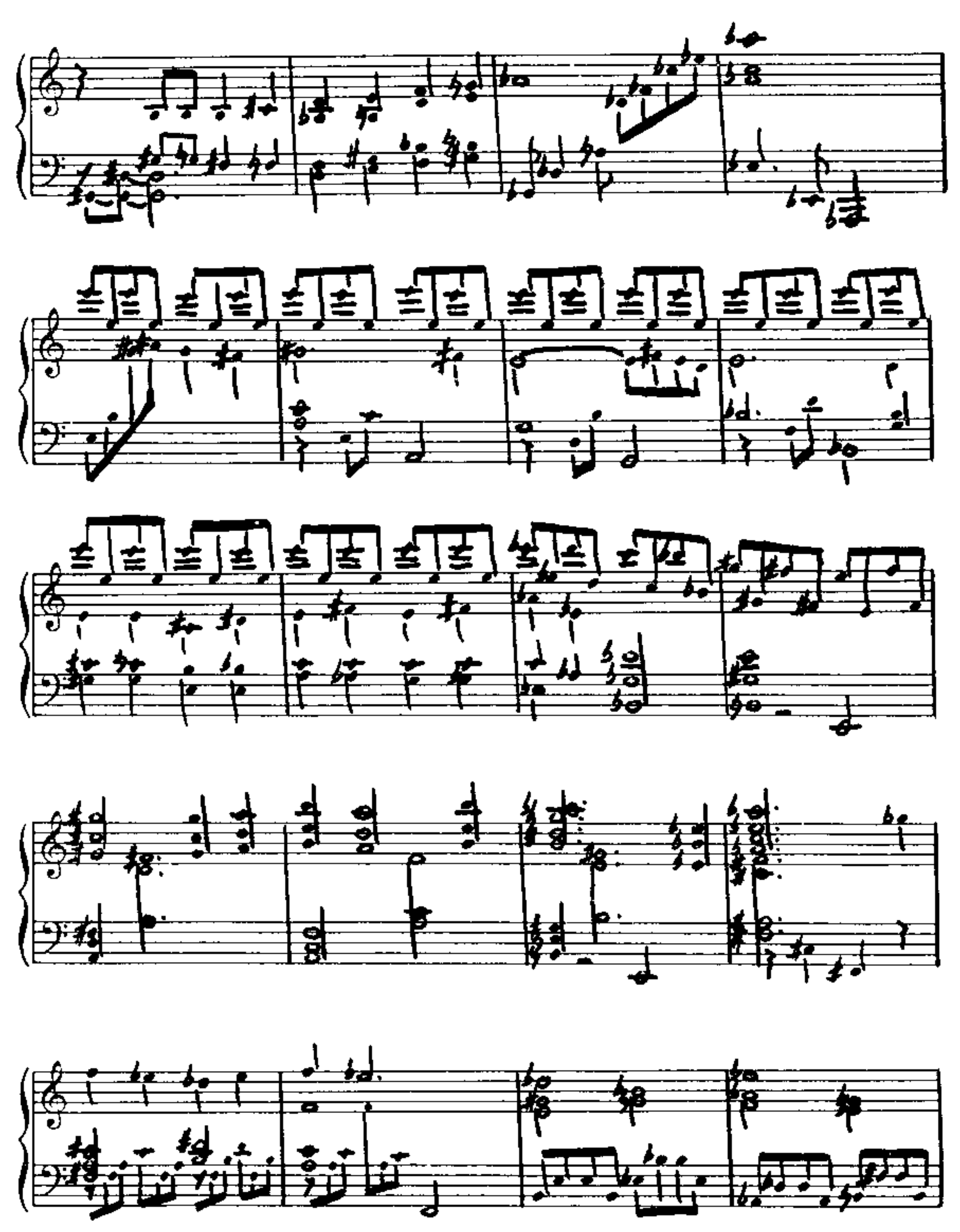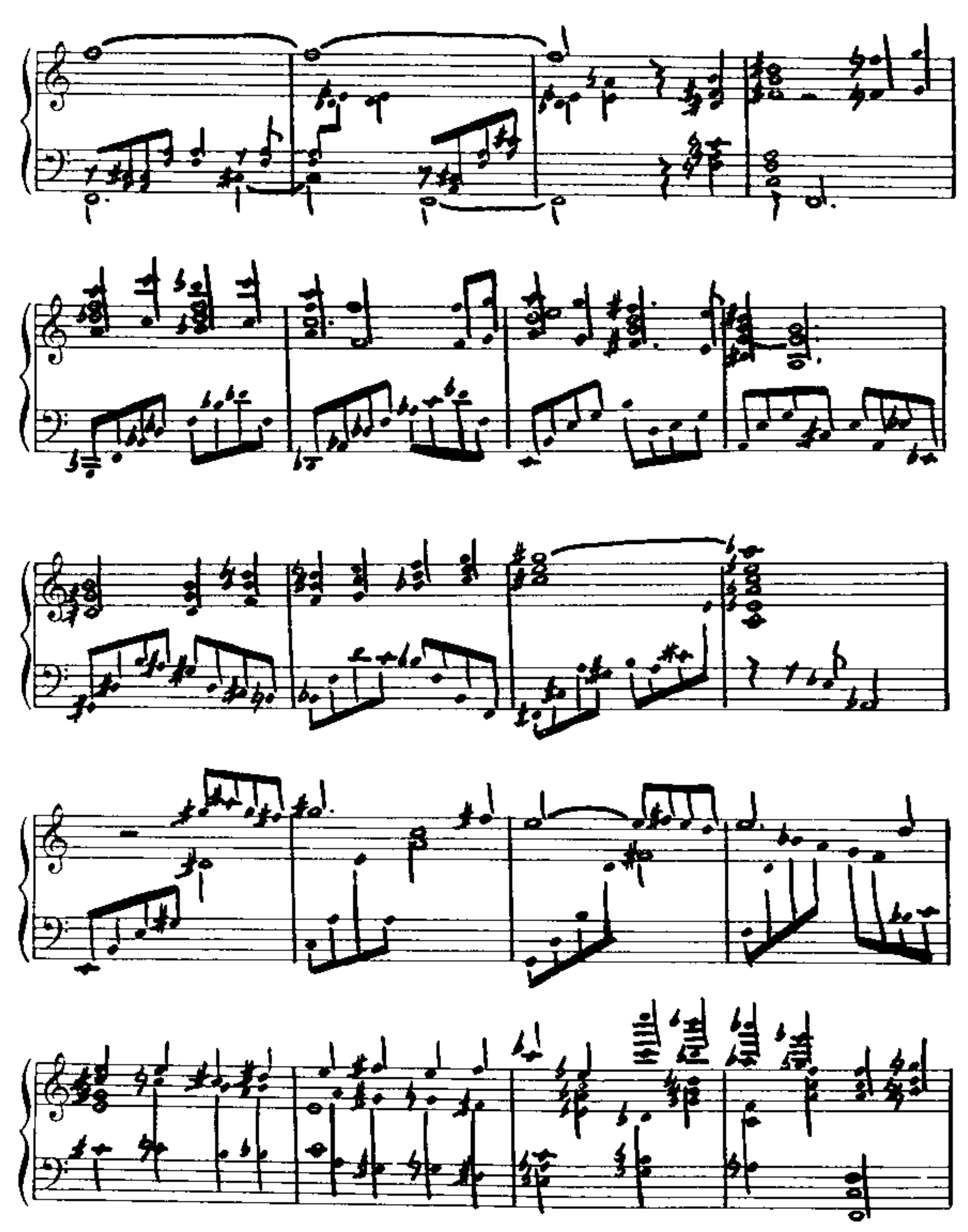




♩ = 76
III
8ba
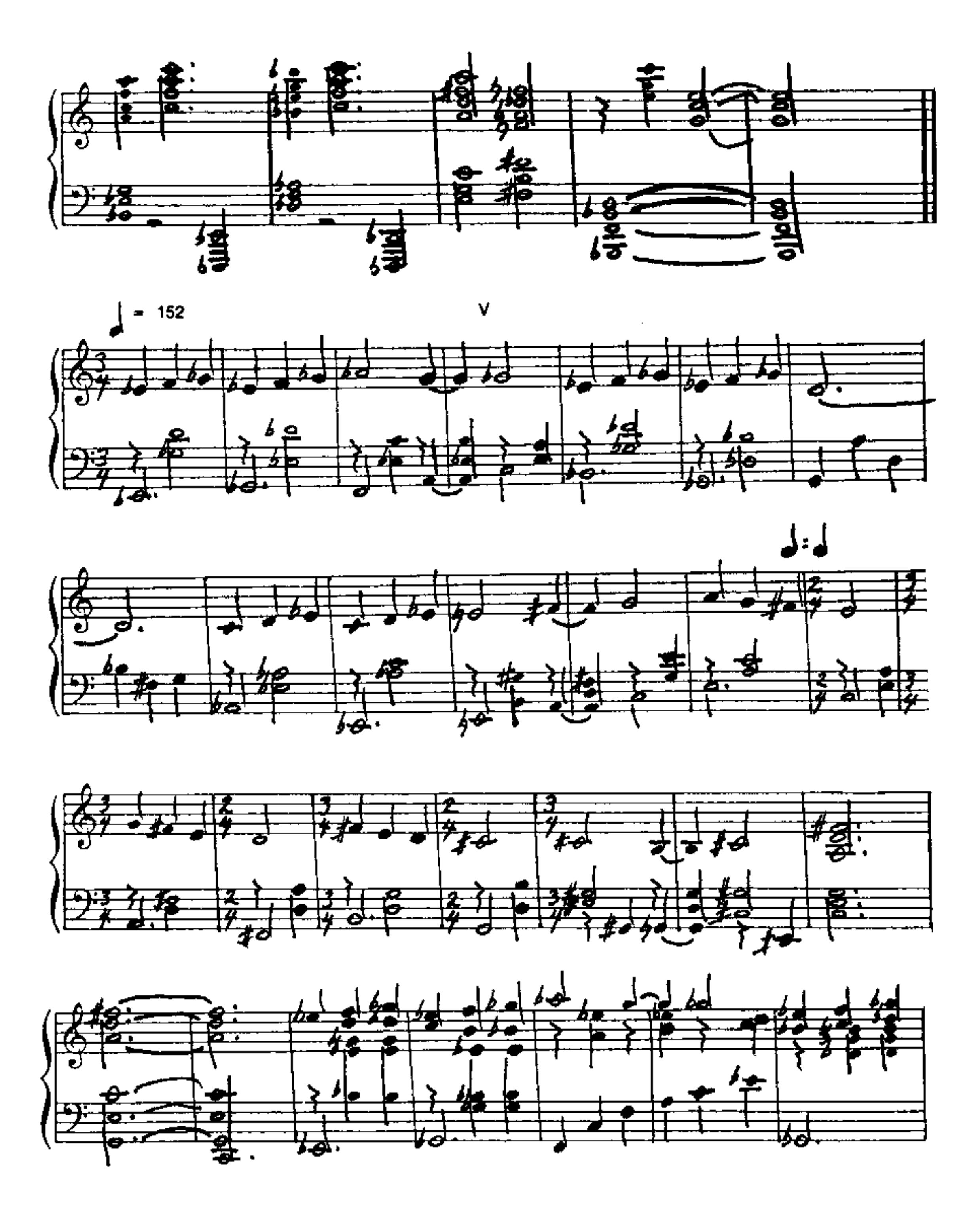

3
3
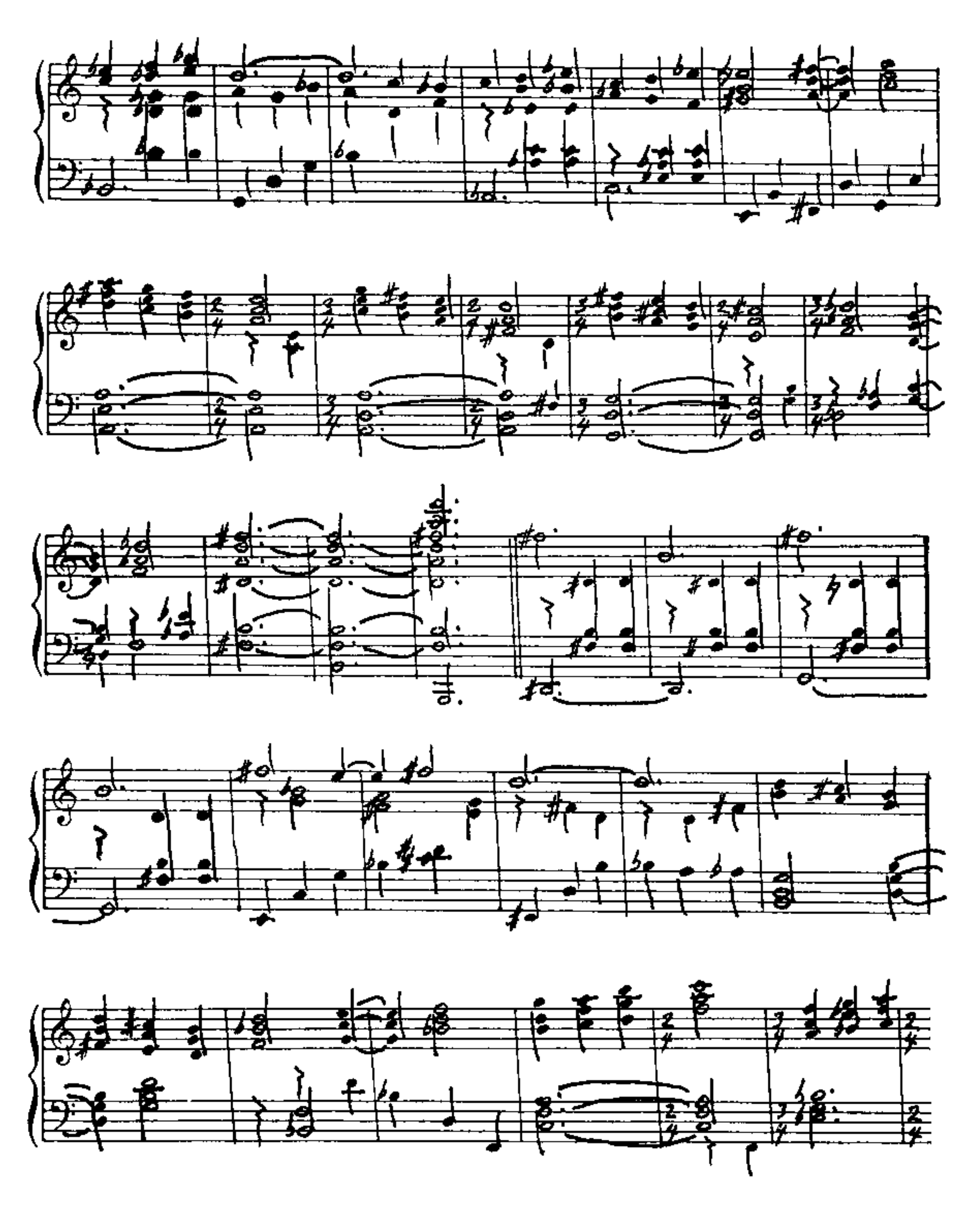


♩ = 120
IV
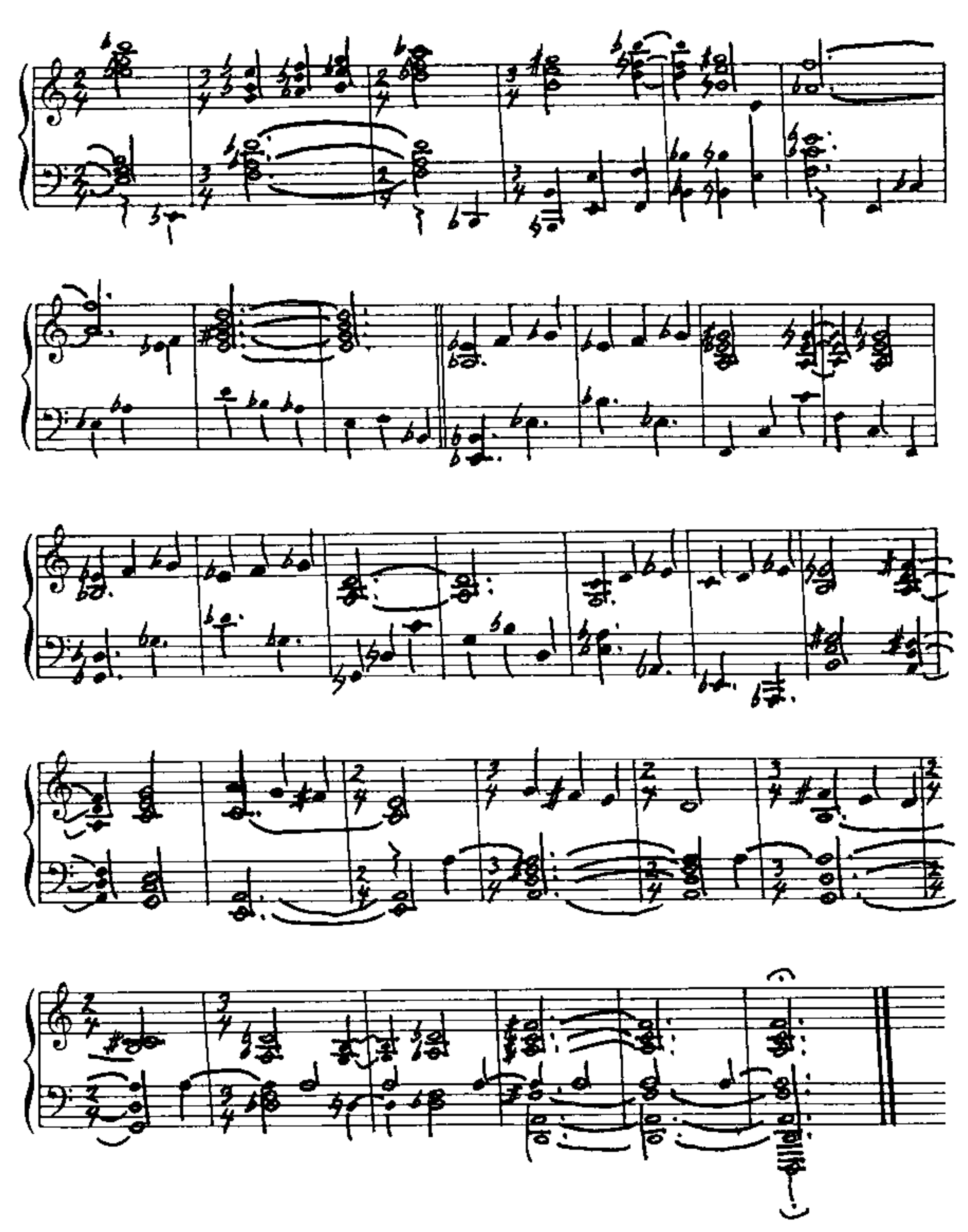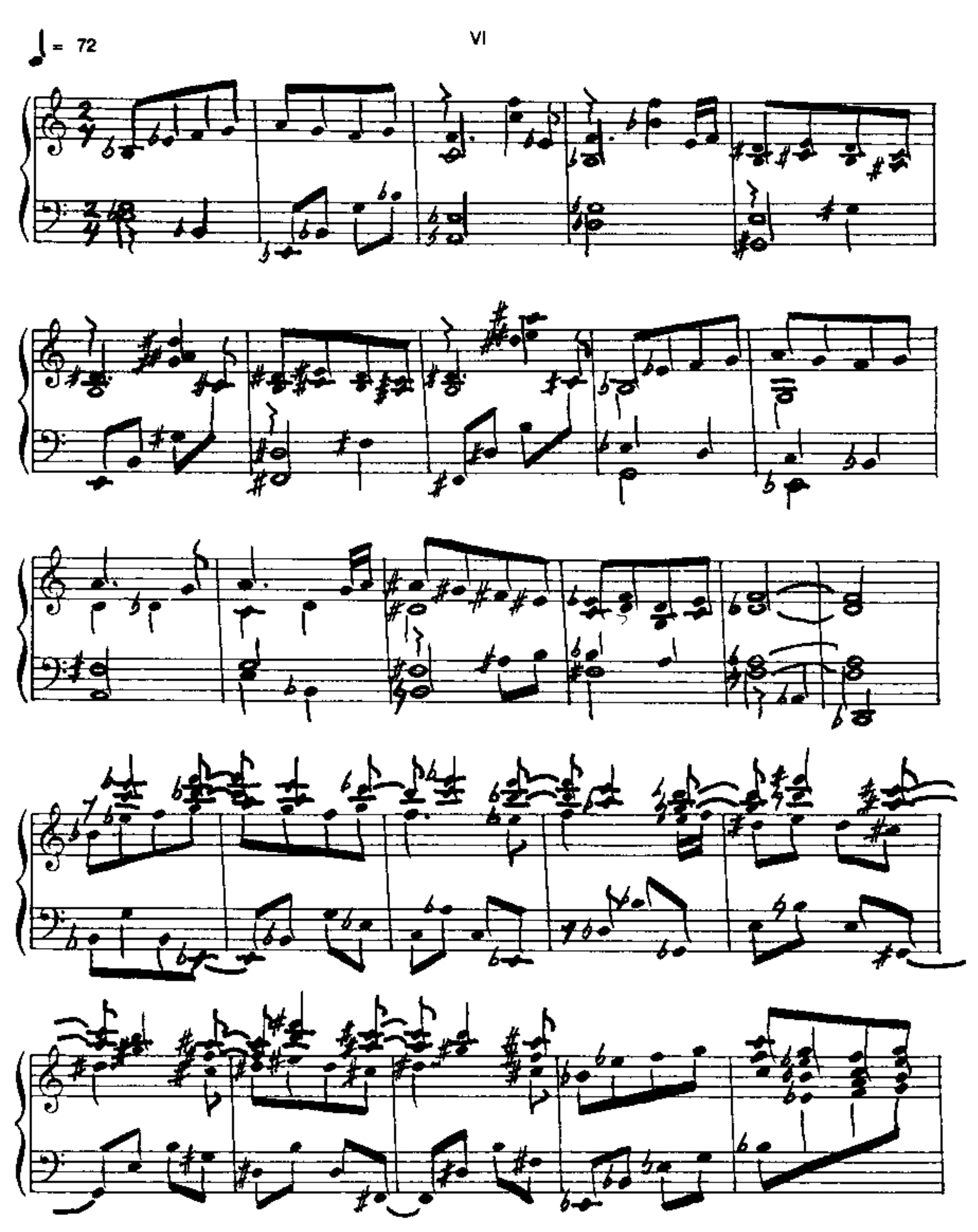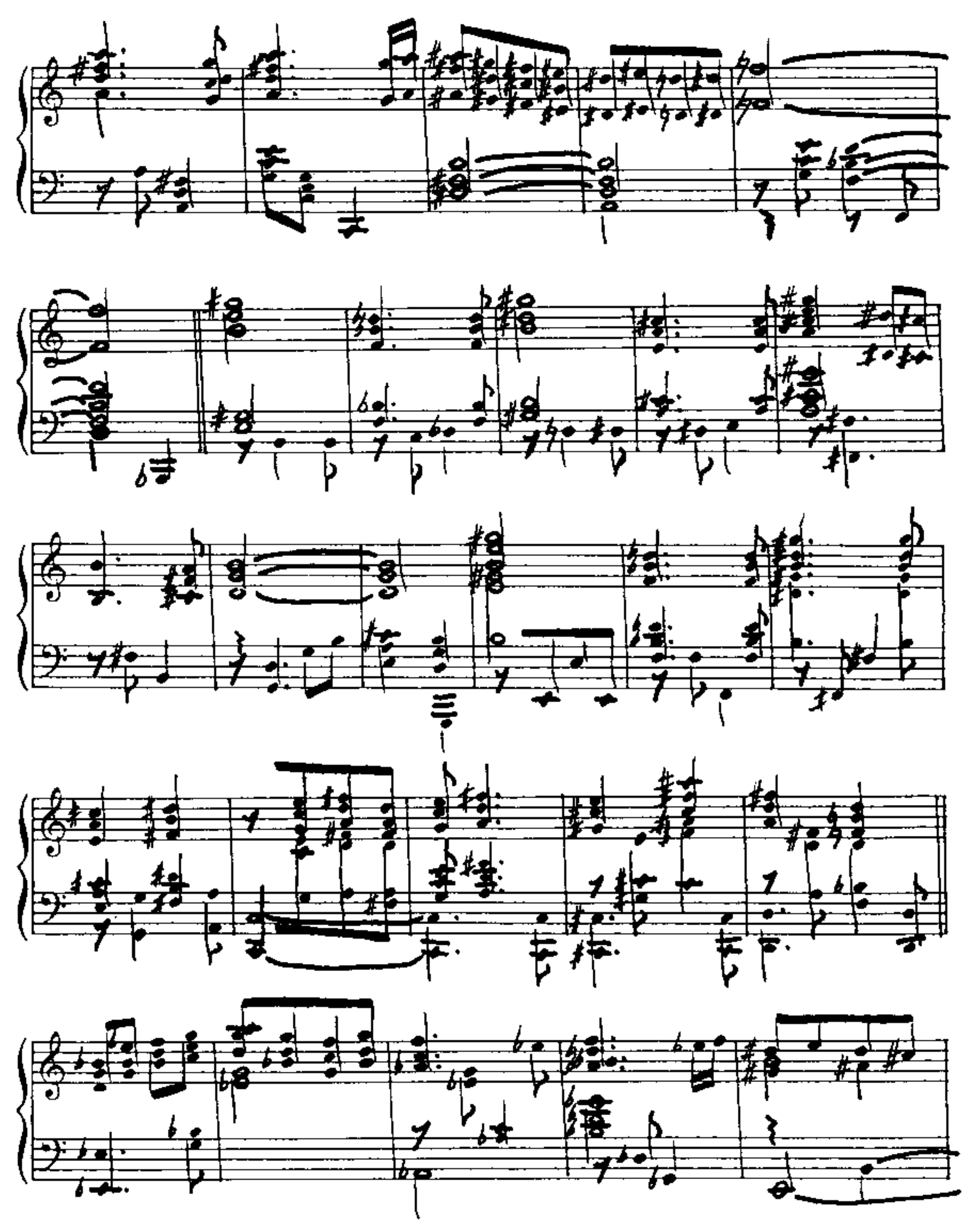






♩ = 152
V
♩ : ♩
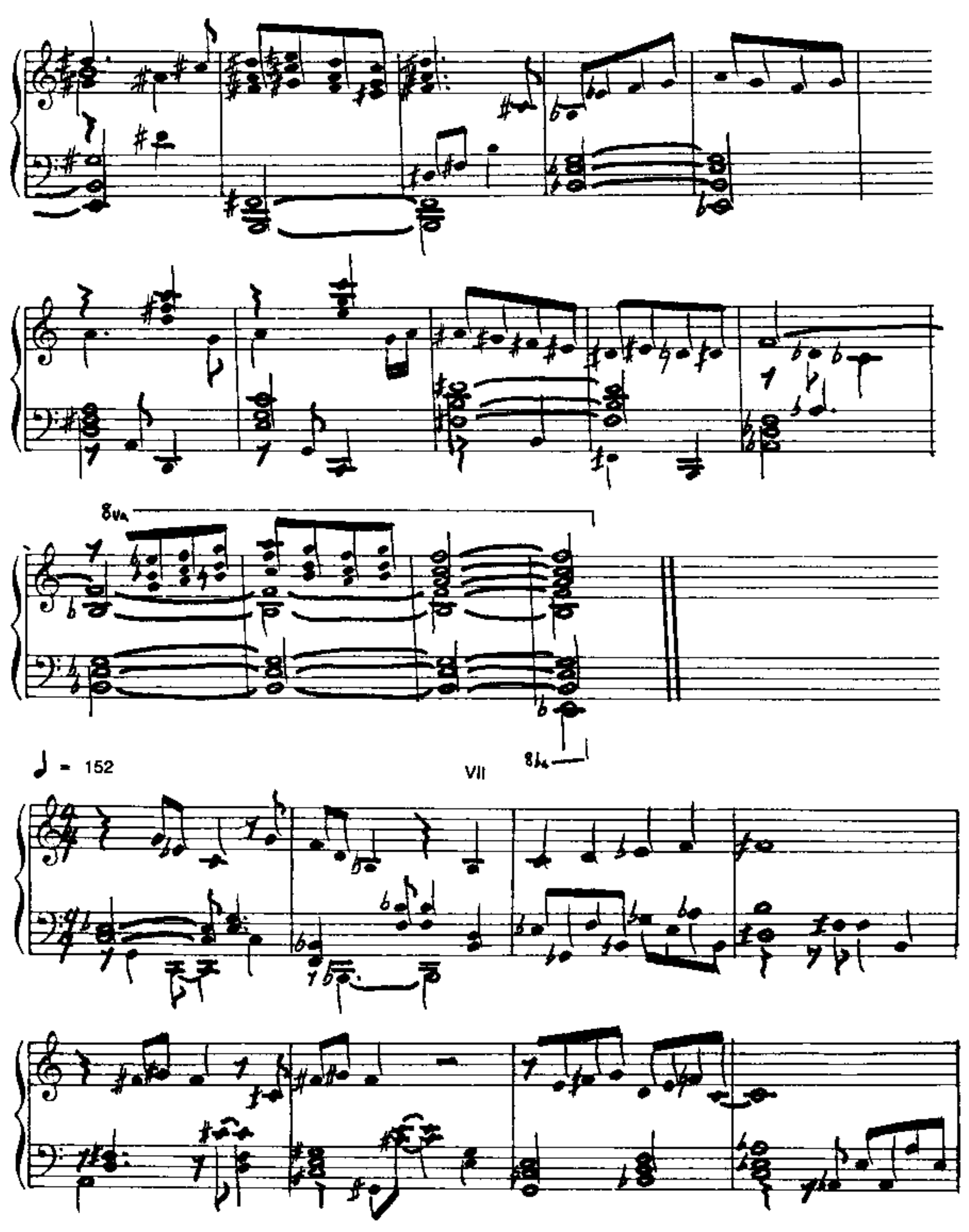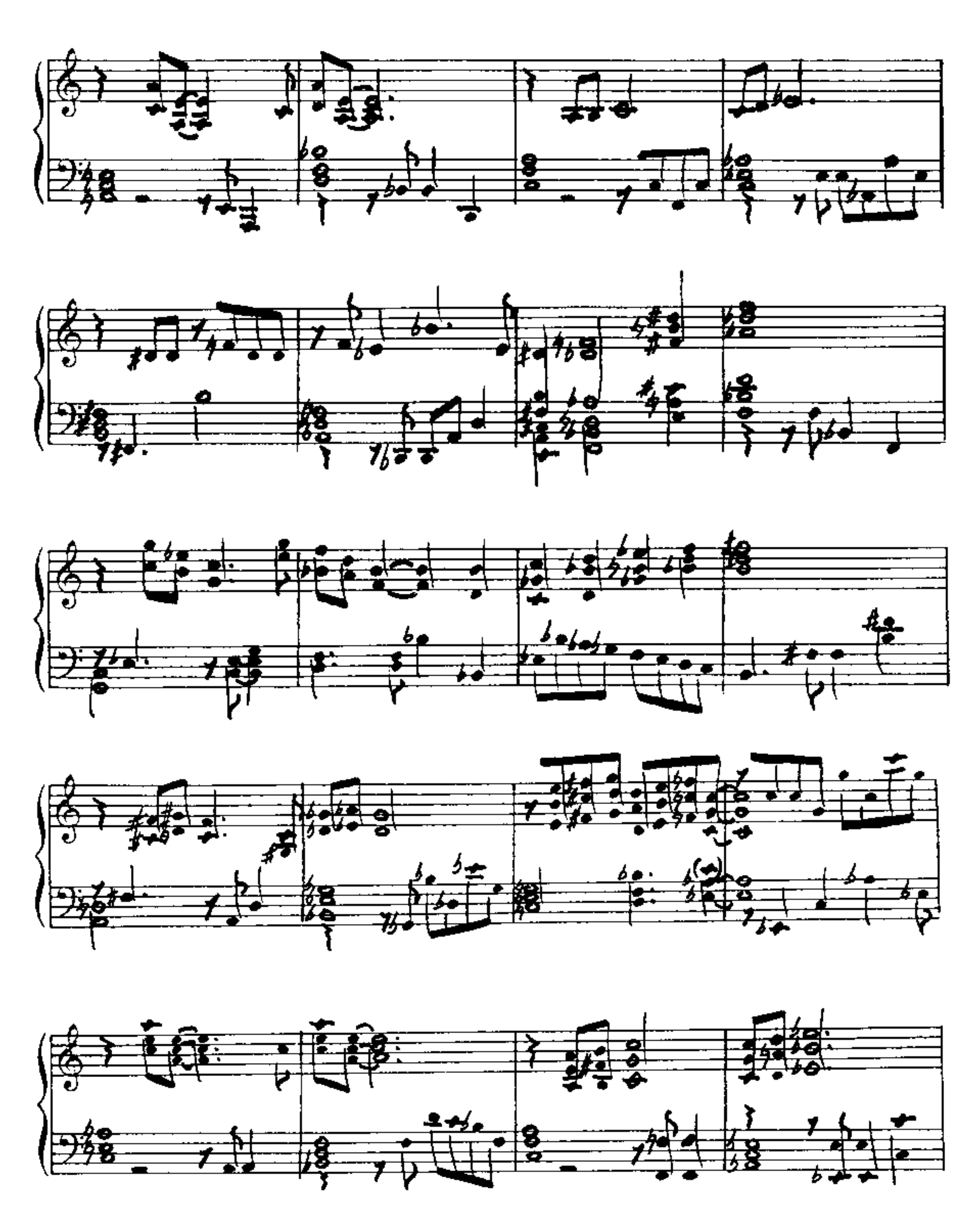




VI
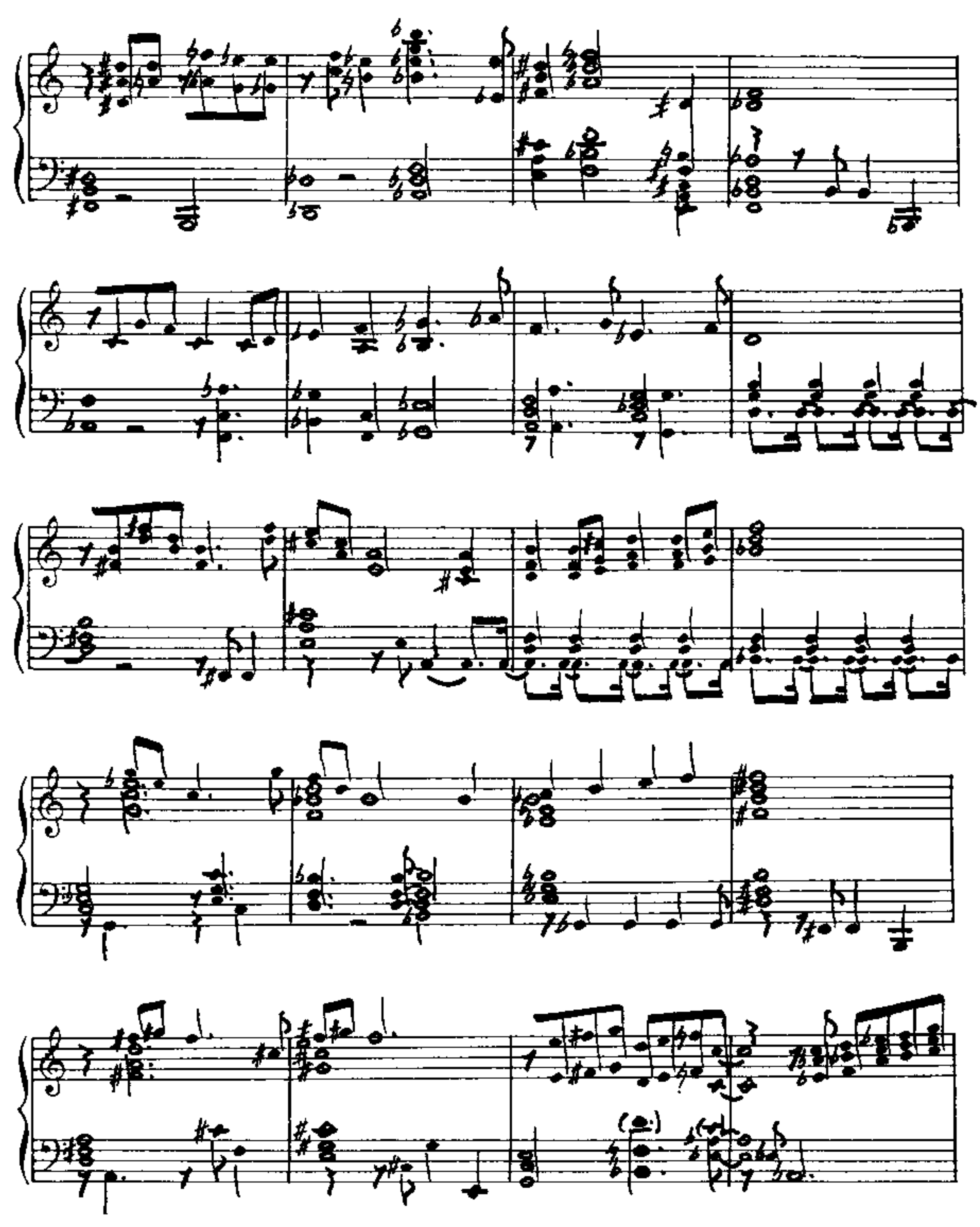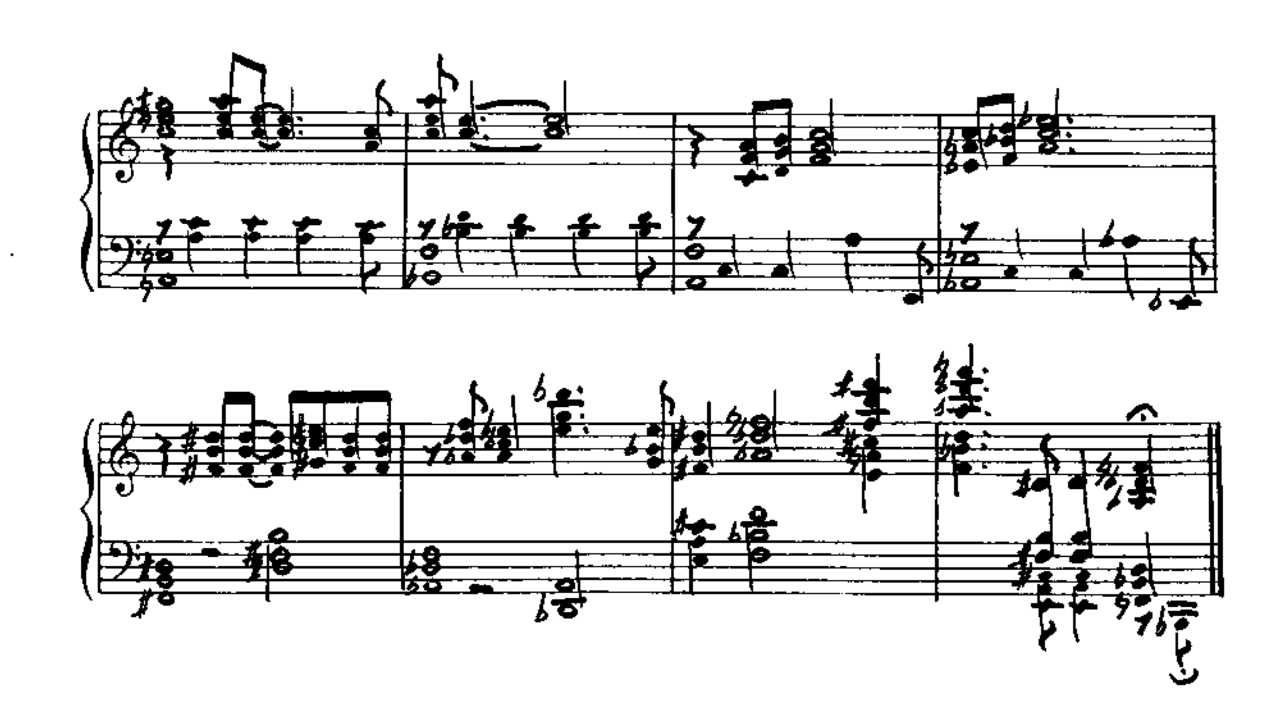






8va
8ba
♩ = 152
VII







Appendix:

The Harmonization/ Melodization Workspace

THE MUSIC FUNCTIONS

Calling Form:	**ADJUST I** Available at S-time and C-time
Right Argument:	I-time index
Result and Purpose:	Resets row I of TGM to the current TM. This is done automatically at the close of S- and C-times. The user need only use this routine to force an earlier or later row of TGM to repeat the current target melody.
Global/Subglobal Variables	
Used:	TGM, TM
Changed:	TGM, TM
Functions Called:	FITO, IROW
This Function Called By:	HOFMIN
Listed on Page:	328
Calling Form:	**F ALL P** Available at C-time

Left Argument: Numeric vector of chordal functions (odd integers from 1 to 13)

Right Argument: Numeric pitch vector

Result and Purpose: Retains in HA only those structures for which all P are in some HA[;FN F] or all HA[;FN F] contain in entries in P.

Global Subglobal Variables Used: HA

Functions Called: NARROW, UNIQ

This Function Called By: User

Listed on Page: 328

Calling Form: **X←Y APND Z**

Left Argument: Scalar, vector or two-dimensional array

Right Argument: Scalar, vector or two-dimensional array. (Must be same type—numeric or character—as left argument.)

Result and Purpose: Returns matrix X containing the elements of Y followed row-wise by the elements of Z.

Global/Subglobal Variables Used: ⎕IO

Local Variables: A, S, T

This Function Called By: Many other functions as a utility

Listed on Page: 328

Calling Form: **BACK N**

Right Argument: Positive integer

Result and Purpose: Returns to the beginning of "x"-time for the current I, where "x" depends on N:

- N = 0 reinitializes the current "time"
- N = 1 returns to the previous "time"
- N ≥ 2 returns to I-time

Global/Subglobal Variables Used: J (Right argument of HOFMIN denoting current "time.")

Changed: Z (Variable returned by HOFMIN denoting the next "time.")

This Function Called By: User

Listed on Page: 328

Calling Form: **ROWS CHNGTM V**
May be used at S-time or C-time

Left Argument: Numeric scalar or vector indexing the ROWS of TGM to be changed

Right Argument: Numeric scalar or vector of chromatic intervals

Result and Purpose: Entries in the given ROWS of TGM are transposed through the intervals in V. These intervals are used successively within each row. Notice the need to relate the number of entries in V (if a vector) to the number of entries in each row. If ROWS contains the current I-value, the new TM will be displayed.

Global/Subglobal Variables
Used: I, TGM
Changed: TM

Functions Called: IROWTGM

This Function Called By: User

Listed on Page: 328

Calling Form: **CLASS N**
May be used at S-time or C-time

Right Argument: Numeric scalar or vector of class indices

Result and Purpose: Retains in HA or structures (rows) in these classes (N).

Global/Subglobal Variables
Used: CLS, CLSNDX, J
Changed: S

Functions Called: NARROW, SLCT

This Function Called By: User

Listed on Page: 328-329

Calling Form:	**S CNKTM N** May be used at S-time or C-time
Left Argument:	Numeric scalar or empty vector (⍳0). This defines the starting pitch. If empty, the first pitch of TM is used.
Right Argument:	Numeric scalar denoting the largest interval allowed between pitches in S, TM (should be greater than half an octave).
Result and Purpose:	Resets the octave of any pitch in TM that would otherwise skip an interval greater than N in S, TM
Global/Subglobal Variables Used:	TM, TONSYS
Local Variables:	II, J, K
Functions Called:	INT, UPTM
This Function Called By:	User
Listed on Page:	329

Calling Form:	**DSPLAH** Not a user function but can be called when execution is stopped to examine the current state of the H-continuity.
Result and Purpose:	Displays the established harmonic continuity (the H array) up to the current chord index (I).
Global/Subglobal Variables Used:	CLSNDX, CLSNM, H, SIGMA
Local Variables:	K
Functions Called:	N2NP
This Function Called By:	SHOW
Listed on Page:	329

Calling Form:	**DSTBN** Not a user function
Result and Purpose:	Sets up global "freedom" variables FRD and MORP for each structure in HA.
Global/Subglobal Variables Used:	ATK, H, HA, I, MORP

Changed:	FRD, MORP
Variables:	FRDM, FRDP
This Function Called By:	NARROW
Listed on Page:	329

Calling Form:	**Z← A EPSPROW B** Matrix analog of the APL ϵ-operator
Left Argument:	Two-dimensional array
Right Argument:	Two-dimensional reference array. Both arguments should be the same type (numeric or character) and have the same number of rows.
Result and Purpose:	Returns a two-dimensional logical array, the same size as A, indicating which members of A appear in the corresponding row of B.
Local Variables:	I
This Function Called By:	SETEPS
Listed on Page:	329

Calling Form:	**N EQH II** May be used at S-time or C-time
Left Argument:	Numeric scalar—a chromatic interval
Right Argument:	Numeric scalar or vector of previous chord indices (i.e., rows of H)
Result and Purpose:	HA forced to contain duplicate(s) of H[II;] transposed through N.
Global/Subglobal Variables	
Used:	H, HA, J (right argument of HOFMIN), TONSYS
Changed:	HA, Z (Return variable of HOFMIN)
Local Variables:	K
Functions Called:	NARROW
This Function Called By:	User
Listed on Page:	329-330

Calling Form:	**N EQM K** May be used at S-time or C-time
Left Argument:	Numeric scalar—a chromatic interval
Right Argument:	Numeric scalar—a previous chord index
Result and Purpose:	Current target melody—TM and TGM[I;]—reset to TGM[K;]. Transpose through N
Global/Subglobal Variables	
Used:	TGM, TM
Functions Called:	UPTM
This Function Called By:	User
Listed on Page:	330

Calling Form:	**FITM F** Can be used at C-time only
Right Argument:	Any nonzero indices from 1 to 7 of harmonic structures. FN F must be used explicitly in place of F to express F in terms of odd integers from 1 to 13!
Result and Purpose:	The target melody (TM) is changed to fit the "best" row of HA (based on CND as a function of TNS). "Fit" implies all TM entries not already in this structure are moved to the nearest structure entry.
Global/Subglobal Variables	
Used:	CND, HA, TM, TNS
Changed:	TM
Local Variables	J, H, T, Z
Functions Called:	FITO, FITSCL, HTNSNM
This Function Called By:	User
Listed on Page:	330

Calling Form:	**FITMALL F** Can be used at C-time only
Right Argument:	As for FITM, except that there must be at least as many entries in F as there are unique pitches in TM.

Result and Purpose:	As for FITM, except that all pitches are fitted simultaneously so as to force different pitches in TM to become different F entries in this chord.
Global/Subglobal Variables	
Used:	CND, HA, TM, TNS
Changed:	TM
Local Variables:	H, K, Z
Functions Called:	FITO, FITSCL, UNIQ
This Function Called By:	User
Listed on Page:	330

Calling Form:	**Z←NEW FITO OLD** Not a user function
Left Argument:	Numeric pitch vector
Right Argument:	Numeric pitch vector. Both arguments should have the same number of entries.
Result and Purpose:	Z is returned as a numeric pitch vector in which NEW entries that differ from OLD ones by more than 9 semitones (TONSYS-3) are reset to a closer octave.
Global/Subglobal Variables:	
Used:	TONSYS
Local Variables:	K
This Function Called By:	FITM, FITMALL
Listed on Page:	330

Calling Form:	**Z← A FITSCL S** Not a user function
Left Argument:	Numeric pitch vector with unique entries (no more than 7 entries when using "MT" SIGMA structures).
Right Argument:	Numeric pitch scale with at least as many entries as are in the left argument.
Result and Purpose:	Z is returned as A adjusted to lie within S. That is, the result contains the "nearest" scale entries to the original pitches in A but each A entry has moved to a different entry in S.

Global/Subglobal Variables
Used: TONSYS
Local Variables: FITZ, UNIQ

This Function Called By: FITM, FITMALL

Listed on Page: 330-331

Calling Form: **FITZ**
Not a user function

Result and Purpose: This is part of the FITSCL algorithm for finding a unique fit between a target melody and a given scale.

Global/Subglobal Variables
Used: TGT, Z
Changed: Z
Local Variables: I, J, K

Functions called: UNIQ ROWS

This Function Called By: FITSCL

Listed on Page:

Calling Form: **Z←FN F**

Right Argument: Odd integers from 1 to 13

Result and Purpose: Allows user to refer to chordal functions as 1, 3, 5, etc. (first, third, fifth, . . .). Returned Z gives equivalent column indices in harmonic structures such as H or HA.

This Function Called By: User

Listed on Page: 331

Calling Form: **FRDGRP**
For use at C-time

Result and Purpose: Displays the number of HA structures in each of the six FRD groups. Entries following a minus sign relate the current harmonic alternatives to the previous

chord. Those following a plus sign relate to the next chord. If both preceding and following chords exist, the net freedom change between these two is shown as "NET Δ = . . ."

Global/Subglobal Variables Used:	H, I, MORP
Local Variables:	DST
Functions Called:	APND
This Function Called By:	User
Listed on Page:	331-332

Calling Forms:	**A F2FA B** **A F2FN B** **A F2FS B** These provide function-to-function (**F2F**) transitions with the last character (A, N or S) implying ALL, No, or Some. For use at C-time.
Left Argument:	Scalar or vector of odd integers (1 to 13) representing harmonic functions of the previous chord: H[I – 1; FN A]
Right Argument:	Scalar or vector of odd integers as above but referring to the chord being selected: HA[;FN B]
Result and Purpose:	Rows of HA will be kept for which the specified transitions are obeyed, i.e., **F2FA** implies all of the pitches having chordal functions A in the previous chord must be the same as pitches in chordal functions B in rows of HA. (If A has more entries than B, the "all" condition must be met by B with respect to A.) Similar statements substituting "none" and "some" instead of "all" apply to the other two routines.
Global/Subglobal Variables Used:	H, I
Functions Called:	ALL, NO, SOME
This Function Called By:	User
Listed on Page:	332

Calling Form:	**Z← A GROWS M**

Left Argument: Numeric scalar; ¯1 implies⍒, otherwise use ÷ ⍋
Right Argument: Numeric matrix

Result and Purpose: Returns matrix Z containing the same elements in each row as in the corresponding row of M but arranged in increasing order unless A=¯1, in which case the elements are in decreasing order.

Global/Subglobal Variables
Used: ⎕IO
Local Variables: I, J, K

This Function Called By: SGSCL

Listed on Page: 332

Calling Form: **Z←PF HMNY M**

Left Argument: Numeric vector. All but the last entry are pitches that will be allowed in the chord in a specified position. The last entry is a structure (column) index (1-7) that must contain one of the pitches.
Right Argument: Melodic pitch vector to be harmonized.

Result and Purpose: Returns Z as a matrix of chords (in HA format) that harmonize M and also satisfy the requirements given in the description of the left argument. This gives the ability to say that the harmonizing chord must have its root or its fifth, etc., restricted to certain pitches.

Global/Subglobal Variables
Used: NAMES
Changed: NAMES
Local Variables: F, K, P

This Function Called By: User

Listed on Page: 332

Calling Form: **Z←HMNZ TM**

Right Argument: Melodic pitch vector to be harmonized

Result and Purpose: Returns Z as a matrix of chords (in HA format) that harmonize TM.

Global/Subglobal Variables
Used: GSIG, SIGMA, TONSYS
Local Variables: I, K, M

Functions Called: SGSCL, UNIQ
This Function Called By: HMNY, HOFM

Listed on Page: 332-333

Calling Form: **HOFM**
"The" harmonization function

Result and Purpose: Initializes and guides each step in the harmonization process.

Global/Subglobal Variables
Used: ATK, GSIG, H, SIGMA, TONSYS, ZOUT
Changed: H
Local Variables: CLS,CND,DST,F,FRD,HA,HB,I,MORP,NAMES,P,S, TGM,TM,TNS,XF,XP

Functions Called: APND,HMNZ,HOFMIN,HOFMOUT,MXMUV,NARROW,NOREP,NSHLIZ,SLCT,UNIQ

This Function Called By: User

Listed on Page:

Calling Form: Z←HOFMIN J

Right Argument: An integer designating the current "time": 0 implies I-time, 1 => S-time, 2 => C-time.

Result and Purpose: Accepts user input to the HOFM program. Returns Z as an integer controlling the logical transitions between I-, S-, and C-times in HOFM.

Global/Subglobal Variables
Used: I, TGM, TM
Local Variables: X

Functions Called: ADJUST, SHOW
This Function Called By: HOFM

Listed on Page: 333-334

Calling Form:	HOFMOUT I
Right Argument:	An integer vector of row indices in the H array
Result and Purpose:	Produces a listing of the I rows of TGM and the corresponding root-tones and class-names derived from H. This is called at the conclusion of execution by HOFM to display each segment of the final melody and the root-tone and class name of the chord that harmonizes it.
Global/Subglobal Variables	
Used:	H, NAMES, TGM, ZOUT
Changed:	NAMES, ZOUT
Local Variables:	J
Functions Called:	APND, N2P, SETNAMES
This Function Called By:	HOFM, User
Listed on Page:	334

Calling Form:	HPATTERN
Result and Purpose:	Lets the user describe chordal and melodic rhythms to generate dummy globals (M and ATK) for executing HOFM. This allows the real M to be composed during execution.
Global/Subglobal Variables	
Used:	ATK0
Changed:	ATK0, M0
Local Variables:	J, K, RH, X
Functions Called:	APND
This Function Called By:	User
Listed on Page:	334

Calling Form:	H HTNSNM M
Left Argument:	Numeric vector or array of harmonic structures given as pitch indices (no references to SIGMA)
Right Argument:	Numeric pitch vector or array

Result and Purpose:	Returns the tension of each M entry with respect to each chord.

Shape of H	Shape of M	Shape of result
vector	vector	ρM
array	vector	(1↑ρH), ρM
vector	array	↑ ϕρM
array	array	(1↑ ρH), ϕ ρH

Global/Subglobal Variables Used:	TONSYS
Local Variables:	A, T
This Function Called By:	FITM, PRTL
Listed on Page:	334

Calling Form:	**Z←A IROW B**
Left Argument:	Two-dimensional character array
Right Argument:	Character vector or array
Result and Purpose:	Returns a copy of A with just row I (I = the global ATK index) replaced by B (which may have more than one row).
Global/Subglobal Variables Used:	I
Functions Called:	APND
This Function Called By:	ADJUST, IROWTGM, TGTMEL
Listed on Page:	334

Calling Form:	**I ROWTGM V**
Left Argument:	Any ATK index (an "I"-time value)
Right Argument:	Character vector or array to be inserted in place of TGM[I;].
Result and Purpose:	TGM (the character array representing the entire target melody) has its I-th row replaced by V.
Global/Subglobal Variables Used:	TGM

Changed:	TGM
Functions Called:	IROW
This Function Called By:	CHNGTM, UPTM
Listed on Page:	334

Calling Form:	**A KEPS B** Used at C-time
Left Argument:	Any entity having an element of its shape equal to the number of rows in HA, e.g., TNS, CLS, FRD, HA[;0], etc.
Right Argument:	Any value(s) whose presence in the left argument should require the corresponding row of HA to be retained. If this is a scalar or vector, all entries will be tested against all entries in the left argument regardless of that argument's shape. If this is an array, it must have the same number of rows as the left argument and then the test will be made only on corresponding rows.
Result and Purpose:	HA rows are kept whose corresponding A entry (or entries) is contained in B. For example, to keep only chords in classes 1 or 4, enter **CLS KEPS 1 4**
Local Variables:	GO
Functions Called:	SETEPS
This Function Called By:	User
Listed on Page:	335

Calling Form:	**A KRNG B** Used at C-time
Left Argument:	Any entity having an element of its shape equal to the number of rows in HA, e.g., TNS, CLS, FRD, HA[;0], etc.
Right Argument:	Numeric scalar or two-element vector
Result and Purpose:	HA rows are kept whose corresponding A entry (or entries) is greater than or equal to the first B entry and less than or equal to the last. For example, **TNS KRNG**

100 444 would keep rows with corresponding tensions between 100 and 444.

Functions Called:	NARROW
This Function Called By:	User
Listed on Page:	335

Calling Form:	**Z←MANDH I**
Right Argument:	Scalar or vector of ATK (or H-row) indices
Result and Purpose:	Returns M segments and H root-tones and class names for the I indices.
Global/Subglobal Variables Used:	H, TGM
Local Variables:	K
Functions Called:	APND, N2NP, SETNAMES
This Function Called By:	User
Listed on Page:	335

Calling Form:	**Z←P MFNH H**
Left Argument:	Numeric pitch vector
Right Argument:	Numeric pitch vector forming a seven-note harmonic structure.
Result and Purpose:	Returns harmonic functions (odd integers from 1 to 13) for each P entry in H. Zeroes are returned where the corresponding P is not present in H.
Global/Subglobal Variables Used:	TONSYS
This Function Called By:	MFNI
Listed on Page:	335

Calling Form:	**Z←MFNHA J** Analytic function for use at C-time

Right Argument: Row index (or indices) of HA. The SLCT function can be most useful here, e.g.,
MFNHA 3 2 SLCT FRD
or
MFNHA ((TNS≥10)/TNS) SLCT TNS, etc.

Result and Purpose: Returns the harmonic function of each pitch in the current TM with respect to each HA[J;]

Global/Subglobal Variables Used: HA
Local Variables: H, TGM

Functions Called: MFNI
This Function Called By: User

Listed on Page: 335

Calling Form: **Z–MFNI I**

Right Argument: Row index (or indices) of H array

Result and Purpose: Returns harmonic functions (1 to 13) of TGM[I;] in the corresponding H[I;].

Global/Subglobal Variables Used: CLSNDX, CLSNM, H, TGM
Local Variables: K

Functions Called: APND, MFNH
This Function Called By: MFNHA

Listed on Page: 335

Calling Form: **IC MLDZ IS**
For use at C-time

Left Argument: Numeric vector of chromatic intervals ordered by preference

Right Argument: Numeric vector of scale intervals

Result and Purpose: Melodizes the current ("best") HA using the last pitch of the previous TM and the two given interval arguments. A reference note is selected by moving the last pitch through the given chromatic intervals as

ordered in IC until one is found that places the note in the scale of this HA. If no given interval works, the nearest scale position will be used. The scalar intervals, IS, then determine this melodic segment.

Global/Subglobal Variables
Used: ATK, CND, HA, TGM, TM, TNS, TONSYS
Changed: TM
Local Variables: J, K, N, R, Z

Functions Called: CNKTM, UPTM
This Function Called By: User

Listed on Page: 335

Calling Form: **MXMUV UD**
For use at C-time

Right Argument: ¯1, 0 or 1

Result and Purpose: The rows of the HA array are reordered according to their root-tone motions as measured from the preceding (or following) root. If UD = 1, the order follows decreasing upward movement (largest interval first). If UD = 0, decreasing order in either direction is used. For UD = ¯1, the greatest downward motion comes first and order proceeds by increasing upward motion. (HOFM calls this using UD = 1.) Notice the greatest possible movement is TONSYS ÷ 2 (6 in equal temperament) in either direction.

Global/Subglobal Variables
Used: ATK, H, HA, I, SIGMA, TONSYS
Local Variables: K

Functions Called: NARROW
This Function Called By: User, HOFM

Listed on Page: 336

Calling Form: **NAMENT**
For use at C-time

Result and Purpose: Displays NAMES (root-tone and class), TNS (tension value) and FRD (freedom values) for each row of HA.

Global/Subglobal Variables Used: FRD, NAMES, TNS

This Function Called By: User

Listed on Page: 336

Calling Form: NARROW N
For use at C-time

Right Argument: Indices of rows of HA to be kept

Result and Purpose: Rows of HA not indexed in N are removed as are corresponding elements of CLS, TNS, FRD, etc. CND is recalculated as ⌊/TNS.

Global/Subglobal Variables Used: DSTBN, PRTL, SETNAMES

This Function Called By: All functions that place restrictions on the rows of HA.

Listed on Page: 336

Calling Form: F NO P
For use at C-time

Left Argument: Numeric vector of chordal functions (odd integers from 1 to 13)

Right Argument: Numeric pitch vector

Result and Purpose: Retains only those rows of HA in which no pitches that are in P occupy columns FN F.

Global/Subglobal Variables Used: HA

Functions Called: NARROW

This Function Called By: F2FN

Listed on Page: 336

Calling Form: NOREP
For use at C-time

Result and Purpose:	Eliminates from HA all structures that duplicate any of the last (or next) NDUP chords in H, as well as any that would repeat a root-tone in any of the last (or next) RDUP rows.
Global/Subglobal Variables Used:	ATK, CLSNDX, H, HA, NDUP, RDUP, SIGMA
Local Variables:	K
Functions Called:	NARROW
This Function Called By:	HOFM
Listed on Page:	336

Calling Form:	**NSHLIZ** Initialization for the **HOFM** algorithm
Result and Purpose:	Displays the current settings of globals NPH, GSIG, NDNP and RDUP. Constructs the TGM array (character matrix; each row contains the target melody for one chord. This is also placed in the ZOUT array, which will display the initial TGM as well as the final pairings of TM and H.)
Global/Subglobal Variables Used:	ATK, GSIG, M, NDUP, NPH, RDUP, TGM, ZOUT
Changed:	TGM, ZOUT
Local Variables:	I
Functions Called:	APND
This Function Called By:	HOFM
Listed on Page:	337

Calling Form:	**Z←P NVRT M**
Left Argument:	Reference pitch (a numeric scalar)
Right Argument:	Melodic (numeric) vector
Result and Purpose:	Returns a melodic vector. This is constructed by inverting the interval sequence of M and placing the first pitch at the first (inverted) interval away from P. (An inverted interval is measured in the opposite direction, i.e., 1 2 3 becomes ¯1 ¯2 ¯3.)

Functions Called:	INT
This Function Called By:	User
Listed on Page:	337

Calling Form:	**KN←N2NP A**
Right Argument:	Numeric pitch array
Result and Purpose:	Returns A converted to pitch names without octave designation.
Local Variables:	MM
Functions Called:	N2P
This Function Called By:	User
Listed on Page:	337

Calling Form:	**KN←N2P A**
Right Argument:	Numeric pitch array
Result and Purpose:	Returns A converted to pitch names with octave designation.
Global/Subglobal Variables Used:	MM, NTS, NUM, TONSYS
Local Variables:	FLDS, I, KD, KF, N, O, ORHO
Functions Called:	N2POUT
This Function Called By:	User
Listed on Page:	337

Calling Form:	**N2POUT Z**
Right Argument:	1 or 0
Result and Purpose:	Returns pitch-name arrays for N2P and N2NP. Right argument controls inclusion of octave designation.
Global/Subglobal Variables	

Used:	DGTS, FLDS, FORS, I, KD, KF, KN, O, ORHO
Changed:	KF, KN
This Function Called By:	N2P
Listed on Page:	337-338

Calling Form:	**CY OVRIDC CL** Used at S- or C-time
Left Argument:	Root-tone interval (cycle) vector
Right Argument:	Class-index vector
Result and Purpose:	HA is constructed using these cycles (from previous root) and classes only—regardless of TM! If used at S-time, a blank line must follow immediately. Note: You may need to reset TM after this (UPTM).
Global/Subglobal Variables	
Used:	CLSNDX, I, H, HA, SIGMA, TONSYS
Changed:	HA, Z
Local Variables:	R, S, SR
Functions Called:	NARROW, SLCT
This Function Called By:	User
Listed on Page:	338

Calling Form:	**CY OVRIDS SR** Used at S- or C-time
Left Argument:	Root-tone interval (cycle) vector
Right Argument:	Structure index vector (rows of SIGMA)
Result and Purpose:	HA is constructed using these cycles and structures only—regardless of TM! If used at S-time, a blank line must follow immediately. Note: You may need to reset TM after this (UPTM).
Global/Subglobal Variables	
Used:	I, H, HA, SIGMA, TONSYS
Changed:	HA, Z
Local Variables:	R, S

Functions Called: NARROW
This Function Called By: User

Listed on Page: 338

Calling Form: **PRTL N**
Used at C-time

Right Argument: Vector of column indices of HA (1 to 7, not 0)

Result and Purpose: TNS (tension vector) is recalculated for TM with respect to each HA[;N] and CND is reset to the minimum of these tensions.

Global/Subglobal Variables
Used: HA, TM, TNS, TONSYS
Changed: CND, TNS

Functions Called: HTNSNM, UNIQ
This Function Called By: User

Listed on Page: 338

Calling Form: **TM PRTLTN N**
Used at C-time

Left Argument: Any melodic vector
Right Argument: Vector of column indices of HA (1 to 7, not 0)

Result and Purpose: Same as PRTL but uses this TM instead of the current one! New TNS vector is then displayed.

Global/Subglobal Variables
Used: TNS

Functions Called: PRTL
This Function Called By: User

Listed on Page: 338

Calling Form: **A PRTNMS B**
Used at C-time

Left Argument: Any vector (or array) with an entry (or column) for

	each row of HA: CLS, TNS, FRD, etc.
Right Argument:	Any numeric scalar or vector (or 2-row array if A is FRD and has two rows)
Result and Purpose:	Produces a **NAMENT**-type display (NAMES, TNS, FRD) but shows only those rows of HA for which the corresponding A entry is contained in B.
Local Variables:	GO
Functions Called:	SETEPS
This Function Called By:	User
Listed on Page:	338

Calling Form:	**RESETM I**
Right Argument:	One or two valid ATK indices
Result and Purpose:	TGM rows from minimum I to maximum I are reset to their original M entries. Thus changes made with **UPTM** can be "erased."
Functions Called:	TGTMEL
This Function Called By:	User
Listed on Page:	338

Calling Form:	**SETEPS**
Result and Purpose:	Isolates the (A∊B) requirements of **KEPS** and **PRTNMS** and also performs the necessary form of "output," either calling **NARROW** to restrict HA or printing NAMES, TNS, and FRD.
Global/Subglobal Variables	
Used:	A, B, GO
Changed:	B
Local Variables:	Z
Functions Called:	EPSROW, NARROW
This Function Called By:	KEPS, PRTNMS
Listed on Page:	338-339

Calling Form:	**Z–SETNAMES H**
Right Argument:	Array of harmonic structures in HA format
Result and Purpose:	Returns array of (Root-tone, Class name) for each row of the given H
Global/Subglobal Variables Used:	CLSNDX, CLSNM, SIGMA
Local Variables:	K
This Function Called By:	MANDH, NARROW, OVRIDC, OVRIDH, HOFMOUT
Listed on Page:	339

Calling Form:	**Z–R SGSCL S**
Left Argument:	Numeric scalar (a root-tone)
Right Argument:	Vector or array of harmonic interval structures such as rows of SIGMA.
Result and Purpose:	Returns scale(s) in "key" of R with "tonality" S.
Global/Subglobal Variables Used:	TONSYS
Functions Called:	GROWS
This Function Called By:	HMNZ
Listed on Page:	339

Calling Form:	**A SHOW X** Part of the HOFM algorithm, frequently changed to account for different output devices being used (typewriter, offline printer, CRT display, etc.)
Left Argument:	Rows of H constructed so far
Right Argument:	Expression to be executed with any explicit results displayed
Result and Purpose:	Primarily allows user to type only "□ " to see the current setting of the H array up to the last row (I–1) used. Preceding any other expression with "□"

	assures its results will be displayed on the alternate output device.
Functions Called:	DSPLAH
This Function Called By:	HOFMIN
Listed on Page:	339

Calling Form:	**SKIP**
Result and Purpose:	Allows the current H[I;] to be set empty (skipped) leaving this segment of M unharmonized.
Global/Subglobal Variables	
Used:	H, I, SIGMA, TGM
Changed:	H, I, TM, Z
This Function Called By:	User
Listed on Page:	339

Calling Form:	**Z←A SLCT B**
Left Argument:	Vector of preferred values in order of preference
Right Argument:	Vector of entities that might contain the A values, e.g., TNS, FRD[1;], HA[;0], etc.
Result and Purpose:	Returns only those indices of B that address entries in A and these indices are ordered as in A.
Local Variables:	K
This Function Called By:	CLASS, CYCLE, FRDSRT, HOFM, OVRIDC, STRUC
Listed on Page:	339

Calling Form:	**F SOME P** Used at C-time
Left Argument:	Odd integers from 1 to 13
Right Argument:	Numeric pitch vector
Result and Purpose:	HA rows are kept in which any of the P entries appear

in any of the HA[;FN F] entries.

Global/Subglobal Variables
Used: HA

Functions Called: NARROW
This Function Called By: User, F2FS

Listed on Page: 339

Calling Form: **STRUC N**
Used at S- and C-time

Right Argument: Row indices of SIGMA array

Result and Purpose: Restricts HA to the structures indexed in SIGMA by N.

Global/Subglobal Variables
Used: J, HA
Changed: S

Functions Called: NARROW, SLCT
This Function Called By: User

Listed on Page: 340

Calling Form: **MN TGTMEL MX**

Left Argument: An "I" value (an index of ATK or a row of TGM)
Right Argument: Another (higher) "I" value

Result and Purpose: Sets (or resets) the global character array TGM, so that all rows from MN to MX contain the proper entries extracted from M by the ATK vector. (This is set up as a character array so that blanks can be used to fill out what appear to be numeric rows.)

Global/Subglobal Variables
Used: ATK, M, TGM
Changed: TGM
Local Variables: I

Functions Called: IROW

This Function Called By: HOFM, RESETM

Listed on Page: 340

Calling Form: Z←UNIQROWS A

Right Argument: Two-dimensional array (numeric or character)

Result and Purpose: Returns A without any replicated rows.

This Function Called By: FITZ

Listed on Page: 340

Calling Form: UPTM V

Right Argument: Numeric pitch vector

Result and Purpose: Updates TM and TGM[I;] to contain V.

Global/Subglobal Variables
Used: I, P, TGM, TM
Changed: P, TM

Functions Called: IROWTGM
This Function Called By: CNKTM, EQM, MLDZ

Listed on Page: 340

VARIABLES

```
A7       :    scalar   (numeric)
   23

A7F9     :    scalar   (numeric)
   22

A7S9     :    scalar   (numeric)
   24

CLSNDX   :    ρ= 36    (numeric)
   0 0 1 1 1 2 2 3 3 3 3 4 4 4 5 5 5 6 6 6 6 6 7 7 7 8 8 8 8 9 9
   9 10 11 11 11

CLSNM    :    ρ= 12 6   (character)
   -MI7
```

```
   -MA7
   MI7
   MI
   +MI
   LG7
   MA7
   +7
   +MA7
   7+3
   MA7+3
   +MA7+3

DFLTGSIG:    ρ= 36     (numeric)
   17 15 7 5 0 25 30 32 20 16 9 6 3 1 8 12 18 26 27 34 2 11 33
   10 21 28 35 13 4 23 14 29 24 31 19 22

DGTS    :    ρ= 13     (character)
   0 123456789.¯

FORS    :    ρ= 2      (character)
   b#


GSIG    :    ρ= 36     (numeric)
   17 15 7 5 0 25 30 32 20 16 9 6 3 1 8 12 18 26 27 34 2 11 33
   10 21 28 35 13 4 23 14 29 24 31 19 22

L7      :    scalar    (numeric)
   15

L7A3    :    scalar    (numeric)
   30

L7C     :    ρ= 3      (numeric)
   5 7 9

L7F9    :    scalar    (numeric)
   14

L7S9    :    scalar    (numeric)
   16

MA      :    scalar    (numeric)
   17

MAC     :    ρ= 3      (numeric)
   6 8 10

MA7     :    scalar    (numeric)
   17

MA7A5   :    scalar    (numeric)
   19

MI      :    scalar    (numeric)
   7
```

```
MIC       :    ρ= 2      (numeric)
   3 2

MI7       :    scalar    (numeric)
   5

MORP      :    ρ= 2      (logical)
   1 0

MXC       :    ρ= 4      (numeric)
   0 1 4 11

NDUP      :    scalar    (numeric)
   3

NTS       :    ρ= 7      (character)
   CDEFGAB

NUM       :    ρ= 7      (numeric)
   0 2 4 5 7 9 11

RDUP      :    scalar    (numeric)
   2

SHRP      :    scalar    (character)
   #

SIGMA     :    ρ= 36 6    (numeric)
     3 3 4 4 3 3
     3 3 4 4 3 4
     3 3 5 3 3 3
     3 3 5 3 3 4
     3 3 5 3 3 5
     3 4 3 4 3 4
     3 4 3 4 4 3
     3 4 4 3 3 4
     3 4 4 3 3 5
     3 4 4 3 4 3
     3 4 4 3 4 4
     3 5 3 3 3 5
     3 5 3 3 4 4
     3 5 3 3 5 3
     4 3 3 3 5 3
     4 3 3 4 4 3
     4 3 3 5 3 3
     4 3 4 3 4 3
     4 3 4 3 4 4
     4 3 4 4 3 2
     4 3 4 4 3 3
     4 3 4 4 3 4
     4 4 2 3 5 1
     4 4 2 4 4 1
     4 4 2 5 3 1
     4 4 3 3 4 4
     4 4 3 3 5 3
     4 4 3 4 3 4
     4 4 3 4 4 3
```

```
     5 2 3 3 3 5
     5 2 3 4 2 5
     5 2 3 5 1 5
     5 2 4 3 2 5
     5 3 3 3 5 3
     5 3 3 4 4 3
     5 3 3 5 3 3

TONSYS  :    scalar   (numeric)
   12

MM      :    scalar   (logical)
   1
```

FUNCTION LISTINGS

```
     ∇ ADJUST I
[1]    TM←(,TM) FITO,⍎TGM[I;]
[2]    TGM←TGM IROW⍕TM
     ∇



     ∇ F ALL P
[1]    NARROW(((ρ,UNIQ P)⌊ρ,F)=+/HA[;,FN F]∊P)/⍳1↑ρHA
     ∇



     ∇ X←Y APND Z;A;S;T
[1]    T←(((2↑(~T)×ρZ)+(1,ρ,Z)×T←2≠ρρZ)×0≠ρ,Z)ρZ
[2]    S←(((2↑(~S)×ρY)+(1,ρ,Y)×S←2≠ρρY)×0≠ρ,Y)ρY
[3]    A←⌈/(¯1↑ρT),¯1↑ρS
[4]    X←(((1↑ρS),A)↑S),[⎕IO]((1↑ρT),A)↑T
     ∇



     ∇ BACK N
[1]    Z←0⌈J-N
     ∇



     ∇ ROWS CHNGTM V;J;K
[1]    J←,ROWS
[2]   L1:K←((ρK)ρV)+K←⍎TGM[J[0];]
[3]    J[0] IROWTGM⍕K
[4]    →(0<ρJ←1↓J)↑L1
[5]    →(I∊ROWS)↓0
[6]    TM←TONSYS|⍎TGM[I;]
[7]    'TM RESET TO ',TGM[I;]
     ∇



     ∇ CLASS N
```

```
[1]   →(2=J)↑L1
[2]   S←N SLCT CLSNDX
[3]   →0
[4]  L1:NARROW N SLCT CLS
    ∇



    ∇ S CNKTM N;II;J;K
[1]   N←N⌈⌊TONSYS÷2
[2]   S←1↑S,K←,TM
[3]  L1:II←INT S,K
[4]   J←(N<|II)/⍳⍴K
[5]   →(0=⍴J)↑L2
[6]   K[J]←K[J]-TONSYS××II[J]
[7]   →L1
[7]  L2:UPTM K
[8]   'TM RESET TO ',⍕TM
    ∇



    ∇ N←DSPLAH;K
[1]   K←~H[⍳I;0]∊¯1,1↑⍴SIGMA
[2]   N←(N2NP((+/K),1)⍴H[K/⍳I;1]),CLSNM[CLSNDX[H[K/⍳I;0]];]
[3]   N←K\[0] N
[4]   K←~K
[5]   N[K/⍳1↑⍴N;]←((+/K),¯1↑⍴N)⍴(¯1↑⍴N)↑'SKIP'
    ∇



    ∇ DSTBN;FRDM;FRDP
[1]   FRDM←FRDP←(1↑⍴HA)⍴0
[2]   →(I=0)↑L1
[3]   →(¯1∊H[I-1;])↑L1
[4]   FRDM←+/~(0 1 ↓HA)∊1↓H[I-1;]
[5]  L1:→(I≥¯1+⍴ATK)↑L2
[6]   →(¯1∊H[I+1;])↑L2
[7]   FRDP←+/~(0 1 ↓HA)∊1↓H[I+1;]
[8]  L2:MORP←0<(+/FRDM),+/FRDP
[9]   FRD←MORP/[0] FRDM,[¯0.5] FRDP
    ∇



    ∇ Z←A EPSROW B;I
[1]   Z←(⍴A)⍴0
[2]   I←0
[3]  L1:Z[I;]←A[I;]∊B[I;]
[4]   →((1↑⍴A)>I←I+1)↑L1
    ∇



    ∇ N EQH II;K
[1]   HA←((⍴,II),¯1↑⍴H)↑H[,II;]
[2]   HA←HA+0, 0 1 ↓(⍴HA)⍴N
```

```
[3]    HA[;K]←TONSYS|HA[;K←1↓⍳¯1↑⍴HA]
[4]    NARROW⍳1↑⍴HA
[5]    →(J=2)↑0
[6]    Z←3
     ∇


     ∇ N EQM K
[1]    UPTM N+(⍴TM)↑⍎,TGM[K;],' '
     ∇


     ∇ FITM F;H;T;Z;J
[1]    H←1↓HA[TNS⍳CND;]
[2]    T←0<H HTNSNM TM
[3]    →(0=+/T)↑0
[4]    Z←T/TM
[5]    J←0
[6]   L1:Z[J]←Z[J] FITSCL H[,F-1]
[7]    →((⍴Z)>J←J+1)↑L1
[8]    Z←Z FITO T/TM
[9]    TM←(T\Z)+(~T)×TM
[10]   'TM RESET TO ',⍕TM
     ∇


     ∇ FITMALL F;H;K;Z
[1]    H←1↓HA[TNS⍳CND;]
[2]    Z←UNIQ TM
[3]    K←Z⍳TM
[4]    Z←Z FITSCL H[,F-1]
[5]    TM←Z[K] FITO TM
[6]    'TM RESET TO ',⍕TM
     ∇


     ∇ Z←NEW FITO OLD;K
[1]    Z←NEW
[2]   L1:K←((|Z-OLD)>TONSYS-3)/⍳⍴Z
[3]    →(0=⍴K)↑0
[4]    Z[K]←Z[K]-TONSYS××Z[K]-OLD[K]
[5]    →L1
     ∇


     ∇ Z←A FITSCL S;I;J;K;M;TGT;R
[1]    A←TONSYS|,A
[2]    →(((⍴A)=⍴UNIQ,A)∧(⍴A)≤⍴S)↑L1
[3]    'FIRST ARGUMENT CANNOT HAVE REDUNDANT ELEMENTS OR MORE ENTRIES
       THAN SECOND ARGUMENT.'
[4]    →0
[5]   L1:J←S∘.-A
```

```
[6]    TGT←(3,⍴A)⍴0
[7]    TGT[0;]←⌊/[0]|J+TONSYS×J≤0
[8]    TGT[2;]←-⌊/[0]|J-TONSYS×J≥0
[9]    TGT[1;]←(~∨/[0] J=0)×(TGT[2;]×TGT[0;]≥|TGT[2;])+TGT[0;]×TGT[0;]
       <|TGT[2;]
[10]   Z←TONSYS|A+TGT[1;]
[11]   →((⍴A)=⍴UNIQ,Z)↑0
[12]   FITZ
[13]   →((⍴A)=⍴UNIQ,Z)↑0
[14]   K←+/|TGT[0 2 ;]
[15]   K←2×K[1]≤K[0]
[16]   Z←TONSYS|A+TGT[K;]
[17]   →((⍴Z)=⍴UNIQ,Z)↑0
[18] L2:R←(1<+/Z∘.=Z)/⍳⍴Z
[19]   J←⍎('⌊⌈')[K÷2],'/R'
[20]   I←(Z[J]=Z)/⍳⍴Z
[21]   I←(¯1↑I),I,1↑I
[22]   I←⍎('⌊⌈')[K÷2],'/(I≠J)/I'
[23]   M←(1 ¯1)[K÷2]+S⍳Z[J]
[24]   Z[I]←TONSYS|S[M×⍎(4 ¯12)[K÷2]↑'M≠⍴S-(M<0)×¯1+⍴S']
[25]   →((⍴Z)=⍴UNIQ,Z)↓L2
     ∇


     ∇ FITZ;ZZ;I;J;K
[1]    ZZ←UNIQROWS(Z∘.=Z)×1+((⍴Z),⍴Z)⍴⍳⍴Z
[2]  L1:J←¯1+(ZZ[0;]≠0)/ZZ[0;]
[3]    →(1≥⍴J)↑L3
[4]    K←,TGT[;J]
[5]    J←(J,J,J)[⍋|K]
[6]    K←Z[J]+K[⍋|K]-TGT[1;J]
[7]  L2:I←~K∊(Z≠Z[1↑J])/Z
[8]    →(0=+/I)↑0
[9]    J←I/J
[10]   K←I/K
[11]   Z[J[0]]←K[0]
[12]   I←J≠J[0]
[13]   →(1≥⍴J)↑L3
[14]   K←I/K
[15]   J←I/J
[16]   →L2
[17] L3:ZZ← 1 0 ↓ZZ
[18]   →(0<1↑⍴ZZ)↑L1
     ∇


     ∇ Z←FN F
[1]    Z←,⌈F÷2
     ∇


     ∇ FRDGRP;DST
[1]    →(∧/0=MORP)↑0
[2]    DST←(⍳6),[0]⍉+/(⍳6)∘.=FRD
[3]    DST←(' ',MORP/'-+'),' ',⍕DST
```

```
[4]    →(∧/MORP)↓L1
[5]    DST←DST APND ' NET Δ = ',⍕+/~H[I-1;]∊H[I+1;]
[6]   L1:DST
     ∇
```

```
     ∇ A F2FA B
[1]    B ALL H[I-1;FN A]
     ∇
```

```
     ∇ A F2FN B
[1]    B NO H[I-1;FN A]
     ∇
```

```
     ∇ A F2FS B
[1]    B SOME H[I-1;FN A]
     ∇
```

```
     ∇ Z←A GROWS M;I;J;K
[1]   ⍝A MUST BE ¯1 FOR ⍒.  ANY OTHER VALUE WILL GIVE ⍋.
[2]    Z←M
[3]    →(1=¯1↑⍴Z)↑0
[4]    K←(~∧/M=⍉(⌽⍴M)⍴M[;⎕IO])/⍳1↑⍴M
[5]   L1:I←K[⎕IO]
[6]    →(∧/M[I;]=M[I;⎕IO])↑L2
[7]    J←⍎'⍋⍒'[¯1=A],⍕M[I;]
[8]    Z[I;]←M[I;J]
[9]   L2:→(0=⍴K←1↓K)↓L1
     ∇
```

```
     ∇ Z←PF HMNY M;P;F;K
[1]    P←¯1↓PF
[2]    F←¯1↑PF
[3]    Z←HMNZ M
[4]    →(0∊⍴Z)↑0
[5]    K←(,Z[;F]∊P)/⍳1↑⍴Z
[6]    Z←Z[K;]
[7]    NAMES←NAMES[K;]
     ∇
```

```
     ∇ Z←HMNZ TM;I;K;M
[1]    I←0
[2]    K←⍴UNIQ TONSYS|TM
[3]    Z←(0,2+¯1↑⍴SIGMA)⍴0
[4]   L1:M←(K=+/(I SGSCL SIGMA[GSIG;])∊TONSYS|TM)/GSIG
[5]    →(0=⍴M)↑L2
[6]    Z←Z,[0] M,TONSYS|+\I,SIGMA[M;]
```

```
[7]  L2:→(TONSYS>I←I+1)↑L1
[8]   →(0∊⍴Z)↑L3
[9]   →0
[10] L3:'NO STRUCTURES!'
    ∇



    ∇ HOFM;I;S;P;F;HB;HA;TNS;TM;XF;XP;CND;DST;CLS;FRD;NAMES;TGM;MORP
[1]   NSHLIZ
[2]   H←((⍴ATK),2+¯1↑⍴SIGMA)⍴¯1
[3]   I←0
[4]  L0:DST←TNS←CLS←⍳FRD←0
[5]   XF←F←1+⍳1+¯1↑⍴SIGMA
[6]   ' ' APND(29⍴'∘○') APND 'I=',(⍕I),' ?'
[7]   →(L0,L0,L0)[HOFMIN 0]
[8]  L1:TM←TONSYS|⍎TGM[I;]
[9]   S←GSIG
[10]  P←TM
[11]  XP←¯1
[12]  ' ' APND 'TM=',TGM[I;]
[13]  'S, P, XP, F OR XF (OR ⎕) ?'
[14]  →(L0,L1,L1,L3)[HOFMIN 1]
[15] L2:HA←HMNZ TM
[16]  →(0∊⍴HA)↑L1
[17]  HB←UNIQ TONSYS|P
[18]  NARROW((∨/HA[;,F]∊HB)∧∧/∼HA[;,XF]∊XP)/⍳1↑⍴HA
[19]  NARROW S SLCT HA[;0]
[20]  NOREP
[21]  →(0<1↑⍴HA)↓L0
[22]  MXMUV 1
[23] L3:' ' APND 'CND=⌊/TNS?  (CND,TNS,HA,H?)'
[24]  →(L0,L1,L2)[HOFMIN 2]
[25]  H[I;]←HA[TNS⍳CND;]
[26]  →((⍴ATK)>I←H[;0]⍳¯1)/L0
[27]  'OK?'
[28]  →(L0,L0,L0)[HOFMIN 0]
[29]  HOFMOUT⍳⍴ATK
[30]  ZOUT
    ∇



    ∇ Z←HOFMIN J;X
[1]   Z←⍳0
[2]  L1:X←⍞
[3]   →(')'=1↑X)↑L3
[4]   →(0=⍴,X)↑L2
[5]   →('⎕'=1↑X)↑L4
[6]   ⍎X
[7]   →L1
[8]  L2:→(0=J)↑0
[9]   →(∧/TM=⍎TGM[I;])↑0
[10]  ADJUST I
[11]  →0
[12] L3:S∆HOFMIN←1+L3
[13]  S∆HOFMIN←⍳0
```

```
[14]   →L1
[15] L4:H[⍳I;] SHOW 1↓X
[16]   →L1
     ∇
```

```
     ∇ HOFMOUT I;J
[1]    J←0
[2]    NAMES←SETNAMES H
[3]   L1:ZOUT←ZOUT APND ' ' APND(1 3 ⍴'M: '),N2P⍎TGM[J;]
[4]    ZOUT←ZOUT APND 'H:       ',NAMES[(,I)[J];]
[5]    →((⍴,I)>J←J+1)/L1
[6]    'TO KEEP H, ENTER: ''NAME←H'', OTHERWISE CR'
[7]   L2:J←⍎⍞
[8]    →(0=⍴,J)↓L2
     ∇
```

```
     ∇ HPATTERN;J;K;X;RH
[1]    ATK0←⍳0
[2]    'ENTER VECTOR OF PAIRS OF (NO. OF H''S, NO. OF T-UNITS/H)'
[3]    J←⎕
[4]    K←(((⍴J)÷2),2)⍴J
[5]    RH←' H B' APND ⍕K
[6]    K←0
[7]   L1:'ENTER ATK PATTERN FOR ',(⍕J[K]),' H''S'
[8]    X←⎕
[9]    ATK0←ATK0,J[K]⍴X
[10]   →((⍴J)>K←K+2)↑L1
[11]   M0←⍎¯1↓,((((⍴ATK0),2)⍴'(⍳'),(⍕((⍴ATK0),1)⍴ATK0),((⍴ATK0),2)⍴'),'
[12]   'DUMMY ATK0 AND M0 NOW SET TO FIT RH:'
[13]   ⍉(10 1 ⍴' ') APND ⍉RH
[14]   ' ' APND (45⍴'-¯-_') APND ' '
     ∇
```

```
     ∇ Z←H HTNSNM M;A;T
[1]    A←TONSYS|⍉M∘.-H
[2]    T←10⊥+/[1] 1 11 2 10 ∘.=A
[3]    Z←T×~∨/[0] A∊0
     ∇
```

```
     ∇ Z←A IROW B
[1]    Z←A[⍳I;] APND B APND((I+1),0)↓A
     ∇
```

```
     ∇ I IROWTGM V
[1]    TGM←TGM IROW V
     ∇
```

```
     ∇ A KEPS B;GO
[1]    GO← 0 1
[2]    SETEPS
     ∇
```

```
     ∇ A KRNG B
[1]    NARROW((A≥1↑B)∧A≤¯1↑B)/⍳⍴A
     ∇
```

```
     ∇ Z←MANDH I;K
[1]    K←0
[2]    Z←''
[3]   L1:Z←Z APND N2NP⍎TGM[(,I)[K];]
[4]    →((⍴,I)>K←K+1)↑L1
[5]    Z←(((Z,' '),':'),' '),SETNAMES H[,I;]
     ∇
```

```
     ∇ Z←P MFNH H
[1]    Z←1+2×(1↓H)⍳TONSYS|P
[2]    Z←Z-15×Z>13
     ∇
```

```
     ∇ Z←MFNHA I;H;TGM
[1]    H←HA[I;]
[2]    TGM←⍕((⍴I),⍴TM)⍴TM
[3]    Z←MFNI⍳⍴I
     ∇
```

```
     ∇ Z←MFNI I;K
[1]    Z←''
[2]   L1:K←''⍴1↑I
[3]    Z←Z APND CLSNM[CLSNDX[H[K;0]];], 3 0 ⍕(⍎TGM[K;]) MFNH H[K;]
[4]    →(0<⍴I←1↓I)↑L1
     ∇
```

```
     ∇ IC MLDZ IS;J;K;N;Z;R
[1]    R←¯1+¯1↑⍴HA
[2]    Z←HA[TNS⍳CND;1+⍳R]
[3]    Z←Z[⍋Z]
[4]    K←¯1↑⍎TGM[I-1;]
[5]    J←((J<⍴Z)/J←Z⍳TONSYS|K+IC),N⍳⌊/N←TONSYS|Z-K
[6]    TM←(-ATK[I])↑Z[R|+\J[0],ATK[I]⍴,IS]
[7]    K CNKTM 6
[8]    UPTM TM
     ∇
```

```
    ∇  MXMUV UD;K
[1]    →(I=0)↑L1
[2]    →(H[I-1;0]∊¯1,1↑⍴SIGMA)↑L1
[3]    K←HA[;1]-H[I-1;1]
[4]    →L2
[5]   L1:→(I≥¯1+⍴ATK)↑0
[6]    →(H[I+1;1]∊¯1,1↑⍴SIGMA)↑0
[7]    K←H[I+1;1]-HA[;1]
[8]   L2:K←((K)⌈TONSYS÷2)×K-TONSYS)+((K<-⌈TONSYS÷2)×K+TONSYS)+K×(|K)≤
       |⌈TONSYS÷2
[9]    →(L3,L4,L5)[1+UD]
[10]  L3:NARROW(⍳⍴K)[⍋K]
[11]   →0
[12]  L4:NARROW(⍳⍴K)[⍒|K]
[13]   →0
[14]  L5:NARROW(⍳⍴K)[⍒K]
    ∇



    ∇ NAMENT
[1]    NAMES,⍕TNS,⍉FRD
    ∇



    ∇ NARROW N
[1]    →(0=⍴,N)↑L3
[2]    NAMES←SETNAMES HA
[3]    PRTL NPH↑1+2×⍳7
[4]    HA←HA[,N;]
[5]    CLS←CLSNDX[HA[;0]]
[6]    NAMES←NAMES[,N;]
[7]    TNS←TNS[,N]
[8]    DSTBN
[9]    CND←⌊/TNS
[10]   →0
[11]  L3:'NO SUCH STRUCTURES (HA UNCHANGED)'
    ∇



    ∇ F NO P
[1]    NARROW(∧/~HA[;,FN F]∊P)/⍳1↑⍴HA
    ∇



    ∇ NOREP;K
[1]    →(0=NDUP)↑L1
[2]    K←((K≥0)∧K<⍴ATK)/K←I+(1+⍳NDUP),-1+⍳NDUP
[3]    K←(~H[K;0]∊¯1,1↑⍴SIGMA)/K
[4]    →(0=⍴K)↑L1
[5]    NARROW(~∨/(CLSNDX[HA[;,0]],HA[;,1])∧.=⍉(CLSNDX[H[K;,0]]),H[K;,
       1])/⍳1↑⍴HA
[6]   L1:→(0=RDUP)↑0
[7]    K←((K≥0)∧K<⍴ATK)/K←I+(1+⍳RDUP),-1+⍳RDUP
[8]    NARROW(~HA[;1]∊H[K;1])/⍳1↑⍴HA
    ∇
```

```
     ∇ NSHLIZ;I
[1]    'NPH: ',⍕NPH
[2]    'GSIG CURRENTLY SET TO ' APND(⍕GSIG) APND ' '
[3]    'RDUP AND NDUP SET TO ',(⍕RDUP),' AND ',⍕NDUP
[4]    I←0
[5]    TGM←''
[6]   L1:TGM←TGM APND⍕M[(-⌈ATK[I])↑⍳⌈(+\ATK)[I]]
[7]    →((ρATK)>I←I+1)↑L1
[8]    ZOUT←' ' APND (45ρ'-¯-_') APND ' '
[9]    ZOUT←ZOUT APND 'ORIGINAL TARGET M:' APND ' ',' ',' ',TGM
[10]   ZOUT←ZOUT APND 29ρ'∘o'
     ∇


     ∇ Z←P NVRT M
[1]    Z←+\P,-INT M
     ∇


     ∇ KN←N2NP A;MM
[1]    MM←0
[2]    KN←N2P A
     ∇


     ∇ KN←N2P A;N;O;KD;KF;FLDS;ORHO;I
[1]    →(0=ρ,A)/0
[2]    N←(¯2↑1,1,ρA)ρA
[3]    KN←TONSYS|N
[4]    KF←(KF×(|KF)=(⌊/|KF←KN∘.-NUM)∘.×(ρNUM)ρ1),TONSYS
[5]    KF←(1=+\+\KF≠0)×KF
[6]    KF←(¯1↓ρKF)ρ(,KF≠0)/,KF
[7]    KF←(KF≠TONSYS)×KF
[8]    KN←NTS[NUM⍳KN-KF]
[9]    FLDS←4+⌈/,|KF
[10]   ORHO←¯1↑ρKN
[11]   KN←((¯1↓ρKN),ORHO×FLDS)ρ(,KN),((ρ,KN),¯1+FLDS)ρ' '
[12]   →MM↓L1
[13]   KD←⌊N÷TONSYS
[14]   O←((¯1↓ρN),1)↑KD
[15]   KD←KD-(O, 0 ¯1 ↓KD)
[16] L1:N2POUT MM
     ∇


     ∇ N2POUT Z
[1]    I←0
[2]    →(L1,L2)[FLDS=4]
[3]   L1:KN[;I+1+FLDS×⍳ORHO]←(1 0 1 \FORS)[(KF=0)+2×KF≥1]
[4]    KF←(×KF)×(KF≠0)×¯1+|KF
[5]    →(L2,L1)[(FLDS-4)>I←I+1]
[6]   L2:KN←' ',KN
[7]    →Z↓0
[8]    KN[;FLDS×⍳¯1↑ρKD]←(1↓DGTS)[|KD]
```

```
[9]   KN[;¯1+FLDS×1↓⍳¯1↑⍴KD]←('¯ +')[1+× 0 1 ↓KD]
[10]  KN←(⍕O),':',(' ',KN)
    ∇



    ∇ CY OVRIDC CL;R;S;SR
[1]   R←H[I-1;1]+CY
[2]   S←CL SLCT CLSNDX
[3]   SR←(,((⍴,R),⍴,S)⍴S),[0.5],⍉((⍴,S),⍴,R)⍴R
[4]   HA←SR[;0],TONSYS|+\SR[;1],SIGMA[SR[;0];]
[5]   NARROW⍳1↑⍴HA
[6]   Z←3
    ∇



    ∇ CY OVRIDS SR;R;S
[1]   R←H[I-1;1]+CY
[2]   S←(,((⍴,R),⍴,SR)⍴SR),[0.5],⍉((⍴,SR),⍴,R)⍴R
[3]   HA←S[;0],TONSYS|+\S[;1],SIGMA[S[;0];]
[4]   NARROW⍳1↑⍴HA
[5]   Z←3
    ∇



    ∇ PRTL N
[1]   CND←⌊/TNS←+/HA[;FN N] HTNSNM UNIQ TONSYS|TM
    ∇



    ∇ TM PRTLTN N
[1]   PRTL N
[2]   TNS
    ∇



    ∇ A PRTNMS B;GO
[1]   GO← 1 0
[2]   SETEPS
    ∇



    ∇ RESETM I
[1]   (⌊/I) TGTMEL ⌈/I
    ∇



    ∇ SETEPS;Z
[1]   →(1=1↑¯2↑1,⍴A)↑L2
[2]   →((1↑⍴A)=1↑¯2↑0,⍴B)↑L1
[3]   B←((1↑⍴A),¯1↑⍴B)⍴B
[4]  L1:Z←∧⌿A EPSROW B
```

```
[5]     →L3
[6]   L2:Z←,A∊B
[7]   L3:→GO/L4,L5
[8]   L4:Z/[0] NAMES,⍕TNS,⍉FRD
[9]     →0
[10]  L5:NARROW Z/⍳⍴Z
    ∇
```

```
    ∇ Z←SETNAMES H;K
[1]    K←~H[;0]∊¯1,1↑⍴SIGMA
[2]    Z←(N2NP K/[0]((1↑⍴H),1)⍴H[;1]),CLSNM[CLSNDX[K/[0] H[;0]];]
[3]    →(0=+/K)↑0
[4]    Z←K\[0] Z
    ∇
```

```
    ∇ Z←R SGSCL S
[1]    Z←TONSYS|+\0,S
[2]    →(2=⍴⍴S)↑L1
[3]    Z←Z[⍋Z]
[4]    →L2
[5]   L1:Z←1 GROWS Z
[6]   L2:Z←TONSYS|R+Z
    ∇
```

```
    ∇ A SHOW X
[1]    →(0≠⍴,X)↑L1
[2]    DSPLAH,⍕A
[3]    →0
[4]   L1:⍎X
    ∇
```

```
    ∇ SKIP
[1]    H[I;]←(1↑⍴SIGMA),(¯1+¯1↑⍴H)⍴0
[2]    I←H[;0]⍳¯1
[3]    Z←0
[4]    TM←⍎TGM[I;]
    ∇
```

```
    ∇ Z←A SLCT B;K
[1]    Z←(+/K<⎕IO+⍴,A)↑⍋K←(,A)⍳B
    ∇
```

```
    ∇ F SOME P
[1]    NARROW(∨/HA[;,FN F]∊P)/⍳1↑⍴HA
    ∇
```

```
     ∇ STRUC N
[1]    →(2=J)↑L1
[2]    S←N
[3]    →0
[4]   L1:NARROW N SLCT HA[;0]
     ∇
```

```
     ∇ MN TGTMEL MX;I
[1]    I←MN
[2]   L1:TGM←TGM IROW⍪M[(-⌈ATK[I])↑⍳⌈(+\ATK)[I]]
[3]    →(MX≥I←I+1)↑L1
     ∇
```

```
     ∇ Z←UNIQROWS A
[1]    Z←(1= 0 0 ⍉+\A∧.=⍉A)/[0] A
     ∇
```

```
     ∇ UPTM V
[1]    P←TM←,V
[2]    I IROWTGM⍪TM
     ∇
```

Index

Cybernetic Music

by Jaxitron

Discover how computers can aid and enhance traditional music composition!

The computerist who has a special interest in music . . . the musician who's looking for a way to use computers to aid in original music composition or generate new arrangements . . . both will find this a fascinating guide to combining the best of both musical and computer worlds!

The focus here is on using the principles of artificial intelligence modeling and expert systems to help the musician compose! This is where you'll find out how to use your computer to keep track of musical decisions that you've already made . . . to project possible outcomes of different musical combinations . . . to generate possible harmonies for a given melody (or vice versa) . . . or to perform such time-consuming chores as transposing and orchestrating. And what the computer does is all within the parameters that *you* have established!

The basis of this computerized music generation system is the concept that rhythmic, harmonic, and melodic elements of music—even musical idiom, style, and form—can be quantified and described in mathematical terms as vectors, arrays, and matrices. These elements can then be manipulated and processed in any way the composer wishes using the powerful functions of the APL programming language. Because APL has the ability to express in a single statement or operation, ideas that would require multiple lines of code in other languages, it lends itself particularly well to the necessities of creative musical composition.

Writing in highly readable and thought-provoking style, the author has included scores of programming and musical examples—such as the "Cybernetic Songbook" . . . examples that will give you hands-on experience dealing with the theoretical concepts of both programming and musical composition. By showing the potential offered by a combination of a computer and the artistic abilities of the musician, this book provides a fascinating starting point for originating your own compositions via computer!

TAB TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

FPT > $18.95

ISBN 0-8306-1856-2

PRICES HIGHER IN CANADA

1845-1085